Happy

Simon Marlow and the Happy developers

Nov 22, 2022

CONTENTS

CHAPTER

ONE

INTRODUCTION

Happy is a parser generator system for Haskell, similar to the tool yacc for C. Like yacc, it takes a file containing an annotated BNF specification of a grammar and produces a Haskell module containing a parser for the grammar.

Happy is flexible: you can have several Happy parsers in the same program, and each parser may have multiple entry points. Happy can work in conjunction with a lexical analyser supplied by the user (either hand-written or generated by another program), or it can parse a stream of characters directly (but this isn't practical in most cases). In a future version we hope to include a lexical analyser generator with Happy as a single package.

Parsers generated by Happy are fast; generally faster than an equivalent parser written using parsing combinators or similar tools. Furthermore, any future improvements made to Happy will benefit an existing grammar, without need for a rewrite.

Happy is sufficiently powerful to parse full Haskell — GHC itself uses a Happy parser.

Happy can currently generate four types of parser from a given grammar, the intention being that we can experiment with different kinds of functional code to see which is the best, and compiler writers can use the different types of parser to tune their compilers. The types of parser supported are:

1. "standard" Haskell 98 (should work with any compiler that compiles Haskell 98).
2. standard Haskell using arrays

   (this is not the default because we have found this generates slower parsers than the *standard* backend).
3. Haskell with GHC (Glasgow Haskell) extensions.

   This is a slightly faster option than the *standard* backend for Glasgow Haskell users.
4. GHC Haskell with string-encoded arrays. This is the fastest/smallest option for GHC users. If you're using GHC, the optimum flag settings are `-agc` (see *Invoking*).

Happy can also generate parsers which will dump debugging information at run time, showing state transitions and the input tokens to the parser.

1.1 Compatibility

Happy is written in Glasgow Haskell. This means that (for the time being), you need GHC to compile it. Any version of GHC >= 6.2 should work.

Remember: parsers produced using Happy should compile without difficulty under any Haskell 98 compiler or interpreter.[1]

[1] With one exception: if you have a production with a polymorphic type signature, then a compiler that supports local universal quantification is required. See *Type Signatures*.

1.2 Reporting Bugs

Any bugs found in Happy should be reported to me: Simon Marlow marlowsd@gmail.com including all the relevant information: the compiler used to compile Happy, the command-line options used, your grammar file or preferably a cut-down example showing the problem, and a description of what goes wrong. A patch to fix the problem would also be greatly appreciated.

Requests for new features should also be sent to the above address, especially if accompanied by patches :-).

1.3 License

Previous versions of Happy were covered by the GNU general public license. We're now distributing Happy with a less restrictive BSD-style license. If this license doesn't work for you, please get in touch.

Copyright 2009, Simon Marlow and Andy Gill. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDERS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

1.4 Obtaining Happy

Happy's web page can be found at http://www.haskell.org/happy/. Happy source and binaries can be downloaded from there.

CHAPTER

TWO

USING HAPPY

Users of Yacc will find Happy quite familiar. The basic idea is as follows:

- Define the grammar you want to parse in a Happy grammar file.
- Run the grammar through Happy, to generate a compilable Haskell module.
- Use this module as part of your Haskell program, usually in conjunction with a lexical analyser (a function that splits the input into "tokens", the basic unit of parsing).

Let's run through an example. We'll implement a parser for a simple expression syntax, consisting of integers, variables, the operators `+`, `-`, `*`, `/`, and the form `let var = exp in exp`. The grammar file starts off like this:

```
{
module Main where
}
```

At the top of the file is an optional module header, which is just a Haskell module header enclosed in braces. This code is emitted verbatim into the generated module, so you can put any Haskell code here at all. In a grammar file, Haskell code is always contained between curly braces to distinguish it from the grammar.

In this case, the parser will be a standalone program so we'll call the module `Main`.

Next comes a couple of declarations:

```
%name calc
%tokentype { Token }
%error { parseError }
```

The first line declares the name of the parsing function that Happy will generate, in this case `calc`. In many cases, this is the only symbol you need to export from the module.

The second line declares the type of tokens that the parser will accept. The parser (i.e. the function `calc`) will be of type `[Token] -> T`, where `T` is the return type of the parser, determined by the production rules below.

The `%error` directive tells Happy the name of a function it should call in the event of a parse error. More about this later.

Now we declare all the possible tokens:

```
%token
      let             { TokenLet }
      in              { TokenIn }
      int             { TokenInt $$ }
      var             { TokenVar $$ }
      '='             { TokenEq }
```

(continues on next page)

(continued from previous page)

```
      '+'             { TokenPlus }
      '-'             { TokenMinus }
      '*'             { TokenTimes }
      '/'             { TokenDiv }
      '('             { TokenOB }
      ')'             { TokenCB }
```

The symbols on the left are the tokens as they will be referred to in the rest of the grammar, and to the right of each token enclosed in braces is a Haskell pattern that matches the token. The parser will expect to receive a stream of tokens, each of which will match one of the given patterns (the definition of the `Token` datatype is given later).

The `$$` symbol is a placeholder that represents the *value* of this token. Normally the value of a token is the token itself, but by using the `$$` symbol you can specify some component of the token object to be the value.

Like yacc, we include `%%` here, for no real reason.

```
%%
```

Now we have the production rules for the grammar.

```
Exp   : let var '=' Exp in Exp  { Let $2 $4 $6 }
      | Exp1                    { Exp1 $1 }

Exp1  : Exp1 '+' Term           { Plus $1 $3 }
      | Exp1 '-' Term           { Minus $1 $3 }
      | Term                    { Term $1 }

Term  : Term '*' Factor         { Times $1 $3 }
      | Term '/' Factor         { Div $1 $3 }
      | Factor                  { Factor $1 }

Factor
      : int                     { Int $1 }
      | var                     { Var $1 }
      | '(' Exp ')'             { Brack $2 }
```

Each production consists of a non-terminal symbol on the left, followed by a colon, followed by one or more expansions on the right, separated by `|`. Each expansion has some Haskell code associated with it, enclosed in braces as usual.

The way to think about a parser is with each symbol having a "value": we defined the values of the tokens above, and the grammar defines the values of non-terminal symbols in terms of sequences of other symbols (either tokens or non-terminals). In a production like this:

```
n   : t_1 ... t_n   { E }
```

whenever the parser finds the symbols `t_1...t_n` in the token stream, it constructs the symbol `n` and gives it the value `E`, which may refer to the values of `t_1...t_n` using the symbols `$1...$n`.

The parser reduces the input using the rules in the grammar until just one symbol remains: the first symbol defined in the grammar (namely `Exp` in our example). The value of this symbol is the return value from the parser.

To complete the program, we need some extra code. The grammar file may optionally contain a final code section, enclosed in curly braces.

```
{
```

All parsers must include a function to be called in the event of a parse error. In the `%error` directive earlier, we specified that the function to be called on a parse error is `parseError`:

```
parseError :: [Token] -> a
parseError _ = error "Parse error"
```

Note that `parseError` must be polymorphic in its return type `a`, which usually means it must be a call to `error`. We'll see in *Monadic Parsers* how to wrap the parser in a monad so that we can do something more sensible with errors. It's also possible to keep track of line numbers in the parser for use in error messages, this is described in *Line Numbers*.

Next we can declare the data type that represents the parsed expression:

```
data Exp
      = Let String Exp Exp
      | Exp1 Exp1
      deriving Show

data Exp1
      = Plus Exp1 Term
      | Minus Exp1 Term
      | Term Term
      deriving Show

data Term
      = Times Term Factor
      | Div Term Factor
      | Factor Factor
      deriving Show

data Factor
      = Int Int
      | Var String
      | Brack Exp
      deriving Show
```

And the data structure for the tokens…

```
data Token
      = TokenLet
      | TokenIn
      | TokenInt Int
      | TokenVar String
      | TokenEq
      | TokenPlus
      | TokenMinus
      | TokenTimes
      | TokenDiv
      | TokenOB
      | TokenCB
 deriving Show
```

… and a simple lexer that returns this data structure.

```
lexer :: String -> [Token]
lexer [] = []
```

(continues on next page)

(continued from previous page)

```
lexer (c:cs)
      | isSpace c = lexer cs
      | isAlpha c = lexVar (c:cs)
      | isDigit c = lexNum (c:cs)
lexer ('=':cs) = TokenEq : lexer cs
lexer ('+':cs) = TokenPlus : lexer cs
lexer ('-':cs) = TokenMinus : lexer cs
lexer ('*':cs) = TokenTimes : lexer cs
lexer ('/':cs) = TokenDiv : lexer cs
lexer ('(':cs) = TokenOB : lexer cs
lexer (')':cs) = TokenCB : lexer cs

lexNum cs = TokenInt (read num) : lexer rest
      where (num,rest) = span isDigit cs

lexVar cs =
   case span isAlpha cs of
      ("let",rest) -> TokenLet : lexer rest
      ("in",rest)  -> TokenIn : lexer rest
      (var,rest)   -> TokenVar var : lexer rest
```

And finally a top-level function to take some input, parse it, and print out the result.

```
main = getContents >>= print . calc . lexer
```

After which we close the final code section:

```
}
```

And that's it! A whole lexer, parser and grammar in a few dozen lines. Another good example is Happy's own parser. Several features in Happy were developed using this as an example.

To generate the Haskell module for this parser, type the command `happy example.y` (where `example.y` is the name of the grammar file). The Haskell module will be placed in a file named `example.hs`. Additionally, invoking the command `happy example.y -i` will produce the file `example.info` which contains detailed information about the parser, including states and reduction rules (see *Info Files*). This can be invaluable for debugging parsers, but requires some knowledge of the operation of a shift-reduce parser.

2.1 Returning other datatypes

In the above example, we used a data type to represent the syntax being parsed. However, there's no reason why it has to be this way: you could calculate the value of the expression on the fly, using productions like this:

```
Term  : Term '*' Factor         { $1 * $3 }
      | Term '/' Factor         { $1 / $3 }
      | Factor                  { $1 }
```

The value of a `Term` would be the value of the expression itself, and the parser could return an integer.

This works for simple expression types, but our grammar includes variables and the `let` syntax. How do we know the value of a variable while we're parsing it? We don't, but since the Haskell code for a production can be anything at all, we could make it a function that takes an environment of variable values, and returns the computed value of the expression:

```
Exp   : let var '=' Exp in Exp  { \p -> $6 (($2,$4 p):p) }
      | Exp1                    { $1 }

Exp1  : Exp1 '+' Term           { \p -> $1 p + $3 p }
      | Exp1 '-' Term           { \p -> $1 p - $3 p }
      | Term                    { $1 }

Term  : Term '*' Factor         { \p -> $1 p * $3 p }
      | Term '/' Factor         { \p -> $1 p `div` $3 p }
      | Factor                  { $1 }

Factor
      : int                     { \p -> $1 }
      | var                     { \p -> case lookup $1 p of
                                          Nothing -> error "no var"
                          Just i  -> i }
      | '(' Exp ')'             { $2 }
```

The value of each production is a function from an environment *p* to a value. When parsing a `let` construct, we extend the environment with the new binding to find the value of the body, and the rule for `var` looks up its value in the environment. There's something you can't do in `yacc` :-)

2.2 Parsing sequences

A common feature in grammars is a *sequence* of a particular syntactic element. In EBNF, we'd write something like `n+` to represent a sequence of one or more `n`s, and `n*` for zero or more. Happy doesn't support this syntax explicitly, but you can define the equivalent sequences using simple productions.

For example, the grammar for Happy itself contains a rule like this:

```
prods : prod                    { [$1] }
      | prods prod              { $2 : $1 }
```

In other words, a sequence of productions is either a single production, or a sequence of productions followed by a single production. This recursive rule defines a sequence of one or more productions.

One thing to note about this rule is that we used *left recursion* to define it — we could have written it like this:

```
prods : prod                   { [$1] }
      | prod prods             { $1 : $2 }
```

The only reason we used left recursion is that Happy is more efficient at parsing left-recursive rules; they result in a constant stack-space parser, whereas right-recursive rules require stack space proportional to the length of the list being parsed. This can be extremely important where long sequences are involved, for instance in automatically generated output. For example, the parser in GHC used to use right-recursion to parse lists, and as a result it failed to parse some Happy-generated modules due to running out of stack space!

One implication of using left recursion is that the resulting list comes out reversed, and you have to reverse it again to get it in the original order. Take a look at the Happy grammar for Haskell for many examples of this.

Parsing sequences of zero or more elements requires a trivial change to the above pattern:

```
prods : {- empty -}            { [] }
      | prods prod             { $2 : $1 }
```

Yes — empty productions are allowed. The normal convention is to include the comment `{- empty -}` to make it more obvious to a reader of the code what's going on.

2.2.1 Sequences with separators

A common type of sequence is one with a *separator*: for instance function bodies in C consist of statements separated by semicolons. To parse this kind of sequence we use a production like this:

```
stmts : stmt                  { [$1] }
      | stmts ';' stmt        { $3 : $1 }
```

If the `;` is to be a *terminator* rather than a separator (i.e. there should be one following each statement), we can remove the semicolon from the above rule and redefine `stmt` as

```
stmt : stmt1 ';'              { $1 }
```

where `stmt1` is the real definition of statements.

We might like to allow extra semicolons between statements, to be a bit more liberal in what we allow as legal syntax. We probably just want the parser to ignore these extra semicolons, and not generate a ``null statement'' value or something. The following rule parses a sequence of zero or more statements separated by semicolons, in which the statements may be empty:

```
stmts : stmts ';' stmt          { $3 : $1 }
      | stmts ';'               { $1 }
      | stmt            { [$1] }
      | {- empty -}     { [] }
```

Parsing sequences of *one* or more possibly null statements is left as an exercise for the reader...

2.3 Using Precedences

Going back to our earlier expression-parsing example, wouldn't it be nicer if we didn't have to explicitly separate the expressions into terms and factors, merely to make it clear that `'*'` and `'/'` operators bind more tightly than `'+'` and `'-'`?

We could just change the grammar as follows (making the appropriate changes to the expression datatype too):

```
Exp   : let var '=' Exp in Exp  { Let $2 $4 $6 }
      | Exp '+' Exp             { Plus $1 $3 }
      | Exp '-' Exp             { Minus $1 $3 }
      | Exp '*' Exp             { Times $1 $3 }
      | Exp '/' Exp             { Div $1 $3 }
      | '(' Exp ')'             { Brack $2 }
      | int                     { Int $1 }
      | var                     { Var $1 }
```

but now Happy will complain that there are shift/reduce conflicts because the grammar is ambiguous — we haven't specified whether e.g. `1 + 2 * 3` is to be parsed as `1 + (2 * 3)` or `(1 + 2) * 3`. Happy allows these ambiguities to be resolved by specifying the precedences of the operators involved using directives in the header[2]:

[2] Users of `yacc` will find this familiar, Happy's precedence scheme works in exactly the same way.

```
...
%right in
%left '+' '-'
%left '*' '/'
%%
...
```

The `%left` or `%right` directive is followed by a list of terminals, and declares all these tokens to be left or right-associative respectively. The precedence of these tokens with respect to other tokens is established by the order of the `%left` and `%right` directives: earlier means lower precedence. A higher precedence causes an operator to bind more tightly; in our example above, because `'*'` has a higher precedence than `'+'`, the expression `1 + 2 * 3` will parse as `1 + (2 * 3)`.

What happens when two operators have the same precedence? This is when the associativity comes into play. Operators specified as left associative will cause expressions like `1 + 2 - 3` to parse as `(1 + 2) - 3`, whereas right-associative operators would parse as `1 + (2 - 3)`. There is also a `%nonassoc` directive which indicates that the specified operators may not be used together. For example, if we add the comparison operators `'>'` and `'<'` to our grammar, then we would probably give their precedence as:

```
...
%right in
%nonassoc '>' '<'
%left '+' '-'
%left '*' '/'
%%
...
```

which indicates that `'>'` and `'<'` bind less tightly than the other operators, and the non-associativity causes expressions such as `1 > 2 > 3` to be disallowed.

2.3.1 How precedence works

The precedence directives, `%left`, `%right` and `%nonassoc`, assign precedence levels to the tokens in the declaration. A rule in the grammar may also have a precedence: if the last terminal in the right hand side of the rule has a precedence, then this is the precedence of the whole rule.

The precedences are used to resolve ambiguities in the grammar. If there is a shift/reduce conflict, then the precedence of the rule and the lookahead token are examined in order to resolve the conflict:

- If the precedence of the rule is higher, then the conflict is resolved as a reduce.
- If the precedence of the lookahead token is higher, then the conflict is resolved as a shift.
- If the precedences are equal, then
  - If the token is left-associative, then reduce
  - If the token is right-associative, then shift
  - If the token is non-associative, then fail
- If either the rule or the token has no precedence, then the default is to shift (these conflicts are reported by Happy, whereas ones that are automatically resolved by the precedence rules are not).

2.3.2 Context-dependent Precedence

The precedence of an individual rule can be overridden, using context precedence. This is useful when, for example, a particular token has a different precedence depending on the context. A common example is the minus sign: it has high precedence when used as prefix negation, but a lower precedence when used as binary subtraction.

We can implement this in Happy as follows:

```
%right in
%nonassoc '>' '<'
%left '+' '-'
%left '*' '/'
%left NEG
%%

Exp   : let var '=' Exp in Exp  { Let $2 $4 $6 }
      | Exp '+' Exp             { Plus $1 $3 }
      | Exp '-' Exp             { Minus $1 $3 }
      | Exp '*' Exp             { Times $1 $3 }
      | Exp '/' Exp             { Div $1 $3 }
      | '(' Exp ')'             { Brack $2 }
      | '-' Exp %prec NEG       { Negate $2 }
      | int                     { Int $1 }
      | var                     { Var $1 }
```

We invent a new token `NEG` as a placeholder for the precedence of our prefix negation rule. The `NEG` token doesn't need to appear in a `%token` directive. The prefix negation rule has a `%prec NEG` directive attached, which overrides the default precedence for the rule (which would normally be the precedence of '-') with the precedence of `NEG`.

2.3.3 The %shift directive for lowest precedence rules

Rules annotated with the `%shift` directive have the lowest possible precedence and are non-associative. A shift/reduce conflict that involves such a rule is resolved as a shift. One can think of `%shift` as `%prec SHIFT` such that `SHIFT` has lower precedence than any other token.

This is useful in conjunction with `%expect 0` to explicitly point out all rules in the grammar that result in conflicts, and thereby resolve such conflicts.

2.4 Type Signatures

Happy allows you to include type signatures in the grammar file itself, to indicate the type of each production. This has several benefits:

- Documentation: including types in the grammar helps to document the grammar for someone else (and indeed yourself) reading the code.
- Fixing type errors in the generated module can become slightly easier if Happy has inserted type signatures for you. This is a slightly dubious benefit, since type errors in the generated module are still somewhat difficult to find.
- Type signatures generally help the Haskell compiler to compile the parser faster. This is important when really large grammar files are being used.

The syntax for type signatures in the grammar file is as follows:

```
stmts   :: { [ Stmt ] }
stmts   : stmts stmt                { $2 : $1 }
    | stmt                      { [$1] }
```

In fact, you can leave out the superfluous occurrence of `stmts`:

```
stmts   :: { [ Stmt ] }
    : stmts stmt                { $2 : $1 }
    | stmt                      { [$1] }
```

Note that currently, you have to include type signatures for *all* the productions in the grammar to benefit from the second and third points above. This is due to boring technical reasons, but it is hoped that this restriction can be removed in the future.

It is possible to have productions with polymorphic or overloaded types. However, because the type of each production becomes the argument type of a constructor in an algebraic datatype in the generated source file, compiling the generated file requires a compiler that supports local universal quantification. GHC (with the `-fglasgow-exts` option) and Hugs are known to support this.

2.5 Monadic Parsers

Happy has support for threading a monad through the generated parser. This might be useful for several reasons:

- Handling parse errors by using an exception monad (see *Handling Parse Errors*).
- Keeping track of line numbers in the input file, for example for use in error messages (see *Line Numbers*).
- Performing IO operations during parsing.
- Parsing languages with context-dependencies (such as C) require some state in the parser.

Adding monadic support to your parser couldn't be simpler. Just add the following directive to the declaration section of the grammar file:

```
%monad { <type> } [ { <then> } { <return> } ]
```

where `<type>` is the type constructor for the monad, `<then>` is the bind operation of the monad, and `<return>` is the return operation. If you leave out the names for the bind and return operations, Happy assumes that `<type>` is an instance of the standard Haskell type class `Monad` and uses the overloaded names for the bind and return operations.

When this declaration is included in the grammar, Happy makes a couple of changes to the generated parser: the types of the main parser function and `parseError` (the function named in `%error`) become `[Token] -> P a` where `P` is the monad type constructor, and the function must be polymorphic in `a`. In other words, Happy adds an application of the `<return>` operation defined in the declaration above, around the result of the parser (`parseError` is affected because it must have the same return type as the parser). And that's all it does.

This still isn't very useful: all you can do is return something of monadic type from `parseError`. How do you specify that the productions can also have type `P a`? Most of the time, you don't want a production to have this type: you'd have to write explicit `return`Ps everywhere. However, there may be a few rules in a grammar that need to get at the monad, so Happy has a special syntax for monadic actions:

```
n  :  t_1 ... t_n          {% <expr> }
```

The `%` in the action indicates that this is a monadic action, with type `P a`, where `a` is the real return type of the production. When Happy reduces one of these rules, it evaluates the expression

```
<expr> `then` \result -> <continue parsing>
```

Happy uses `result` as the real semantic value of the production. During parsing, several monadic actions might be reduced, resulting in a sequence like

```
<expr1> `then` \r1 ->
<expr2> `then` \r2 ->
...
return <expr3>
```

The monadic actions are performed in the order that they are *reduced*. If we consider the parse as a tree, then reductions happen in a depth-first left-to-right manner. The great thing about adding a monad to your parser is that it doesn't impose any performance overhead for normal reductions — only the monadic ones are translated like this.

Take a look at the Haskell parser for a good illustration of how to use a monad in your parser: it contains examples of all the principles discussed in this section, namely parse errors, a threaded lexer, line/column numbers, and state communication between the parser and lexer.

The following sections consider a couple of uses for monadic parsers, and describe how to also thread the monad through the lexical analyser.

2.5.1 Handling Parse Errors

It's not very convenient to just call `error` when a parse error is detected: in a robust setting, you'd like the program to recover gracefully and report a useful error message to the user. Exceptions (of which errors are a special case) are normally implemented in Haskell by using an exception monad, something like:

```
data E a = Ok a | Failed String

thenE :: E a -> (a -> E b) -> E b
m `thenE` k =
   case m of
       Ok a     -> k a
       Failed e -> Failed e

returnE :: a -> E a
returnE a = Ok a

failE :: String -> E a
failE err = Failed err

catchE :: E a -> (String -> E a) -> E a
catchE m k =
   case m of
      Ok a     -> Ok a
      Failed e -> k e
```

This monad just uses a string as the error type. The functions `thenE` and `returnE` are the usual bind and return operations of the monad, `failE` raises an error, and `catchE` is a combinator for handling exceptions.

We can add this monad to the parser with the declaration

```
%monad { E } { thenE } { returnE }
```

Now, without changing the grammar, we can change the definition of `parseError` and have something sensible happen for a parse error:

```
parseError tokens = failE "Parse error"
```

The parser now raises an exception in the monad instead of bombing out on a parse error.

We can also generate errors during parsing. There are times when it is more convenient to parse a more general language than that which is actually intended, and check it later. An example comes from Haskell, where the precedence values in infix declarations must be between 0 and 9:

```
prec :: { Int }
      : int    {% if $1 < 0 || $1 > 9
                  then failE "Precedence out of range"
                  else returnE $1
        }
```

The monadic action allows the check to be placed in the parser itself, where it belongs.

2.5.2 Threaded Lexers

Happy allows the monad concept to be extended to the lexical analyser, too. This has several useful consequences:

- Lexical errors can be treated in the same way as parse errors, using an exception monad.
- Information such as the current file and line number can be communicated between the lexer and parser.
- General state communication between the parser and lexer — for example, implementation of the Haskell layout rule requires this kind of interaction.
- IO operations can be performed in the lexer — this could be useful for following import/include declarations for instance.

A monadic lexer is requested by adding the following declaration to the grammar file:

```
%lexer { <lexer> } { <eof> }
```

where `<lexer>` is the name of the lexical analyser function, and `<eof>` is a token that is to be treated as the end of file.

When using a monadic lexer, the parser no longer reads a list of tokens. Instead, it calls the lexical analysis function for each new token to be read. This has the side effect of eliminating the intermediate list of tokens, which is a slight performance win.

The type of the main parser function is now just `P a` — the input is being handled completely within the monad.

The type of `parseError` becomes `Token -> P a`; that is, it takes Happy's current lookahead token as input. This can be useful, because the error function probably wants to report the token at which the parse error occurred, and otherwise the lexer would have to store this token in the monad.

The lexical analysis function must have the following type:

```
lexer :: (Token -> P a) -> P a
```

where `P` is the monad type constructor declared with `%monad`, and `a` can be replaced by the parser return type if desired.

You can see from this type that the lexer takes a *continuation* as an argument. The lexer is to find the next token, and pass it to this continuation to carry on with the parse. Obviously, we need to keep track of the input in the monad somehow, so that the lexer can do something different each time it's called!

Let's take the exception monad above, and extend it to add the input string so that we can use it with a threaded lexer.

```
data ParseResult a = Ok a | Failed String
type P a = String -> ParseResult a

thenP :: P a -> (a -> P b) -> P b
m `thenP` k = \s ->
   case m s of
       Ok a     -> k a s
       Failed e -> Failed e

returnP :: a -> P a
returnP a = \s -> Ok a

failP :: String -> P a
failP err = \s -> Failed err

catchP :: P a -> (String -> P a) -> P a
catchP m k = \s ->
   case m s of
      Ok a     -> Ok a
      Failed e -> k e s
```

Notice that this isn't a real state monad — the input string just gets passed around, not returned. Our lexer will now look something like this:

```
lexer :: (Token -> P a) -> P a
lexer cont s =
    ... lexical analysis code ...
    cont token s'
```

the lexer grabs the continuation and the input string, finds the next token `token`, and passes it together with the remaining input string `s'` to the continuation.

We can now indicate lexical errors by ignoring the continuation and calling `failP "error message" s` within the lexer (don't forget to pass the input string to make the types work out).

This may all seem a bit weird. Why, you ask, doesn't the lexer just have type `P Token`? It was done this way for performance reasons — this formulation sometimes means that you can use a reader monad instead of a state monad for `P`, and the reader monad might be faster. It's not at all clear that this reasoning still holds (or indeed ever held), and it's entirely possible that the use of a continuation here is just a misfeature.

If you want a lexer of type `P Token`, then just define a wrapper to deal with the continuation:

```
lexwrap :: (Token -> P a) -> P a
lexwrap cont = real_lexer `thenP` \token -> cont token
```

Monadic productions with %lexer

The `{% ... }` actions work fine with `%lexer`, but additionally there are two more forms which are useful in certain cases. Firstly:

```
n  :  t_1 ... t_n          {%^ <expr> }
```

In this case, `<expr>` has type `Token -> P a`. That is, Happy passes the current lookahead token to the monadic action `<expr>`. This is a useful way to get hold of Happy's current lookahead token without having to store it in the monad.

```
n  :  t_1 ... t_n          {%% <expr> }
```

This is a slight variant on the previous form. The type of `<expr>` is the same, but in this case the lookahead token is actually discarded and a new token is read from the input. This can be useful when you want to change the next token and continue parsing.

2.5.3 Line Numbers

Previous versions of Happy had a `%newline` directive that enabled simple line numbers to be counted by the parser and referenced in the actions. We warned you that this facility may go away and be replaced by something more general, well guess what? :-)

Line numbers can now be dealt with quite straightforwardly using a monadic parser/lexer combination. Ok, we have to extend the monad a bit more:

```
type LineNumber = Int
type P a = String -> LineNumber -> ParseResult a

getLineNo :: P LineNumber
getLineNo = \s l -> Ok l
```

(the rest of the functions in the monad follow by just adding the extra line number argument in the same way as the input string). Again, the line number is just passed down, not returned: this is OK because of the continuation-based lexer that can change the line number and pass the new one to the continuation.

The lexer can now update the line number as follows:

```
lexer cont s =
  case s of
    '\n':s  ->  \line -> lexer cont s (line + 1)
    ... rest of lexical analysis ...
```

It's as simple as that. Take a look at Happy's own parser if you have the sources lying around, it uses a monad just like the one above.

Reporting the line number of a parse error is achieved by changing `parseError` to look something like this:

```
parseError :: Token -> P a
parseError = getLineNo `thenP` \line ->
             failP (show line ++ ": parse error")
```

We can also get hold of the line number during parsing, to put it in the parsed data structure for future reference. A good way to do this is to have a production in the grammar that returns the current line number:

```
lineno :: { LineNumber }
        : {- empty -}      {% getLineNo }
```

The semantic value of `lineno` is the line number of the last token read — this will always be the token directly following the `lineno` symbol in the grammar, since Happy always keeps one lookahead token in reserve.

2.5.4 Summary

The types of various functions related to the parser are dependent on what combination of `%monad` and `%lexer` directives are present in the grammar. For reference, we list those types here. In the following types, *t* is the return type of the parser. A type containing a type variable indicates that the specified function must be polymorphic.

- **No ``%monad`` or ``%lexer``**

```
parse      :: [Token] -> t
parseError :: [Token] -> a
```

- **with ``%monad``**

```
parse      :: [Token] -> P t
parseError :: [Token] -> P a
```

- **with ``%lexer``**

```
parse      :: T t
parseError :: Token -> T a
lexer      :: (Token -> T a) -> T a
```

  where the type constructor `T` is whatever you want (usually `T a = String -> a`). I'm not sure if this is useful, or even if it works properly.

- **with ``%monad`` and ``%lexer``**

```
parse      :: P t
parseError :: Token -> P a
lexer      :: (Token -> P a) -> P a
```

2.6 The Error Token

Happy supports a limited form of error recovery, using the special symbol `error` in a grammar file. When Happy finds a parse error during parsing, it automatically inserts the `error` symbol; if your grammar deals with `error` explicitly, then it can detect the error and carry on.

For example, the Happy grammar for Haskell uses error recovery to implement Haskell layout. The grammar has a rule that looks like this:

```
close : '}'                 { () }
      | error             { () }
```

This says that a close brace in a layout-indented context may be either a curly brace (inserted by the lexical analyser), or a parse error.

This rule is used to parse expressions like `let x = e in e'`: the layout system inserts an open brace before `x`, and the occurrence of the `in` symbol generates a parse error, which is interpreted as a close brace by the above rule.

Note for `yacc` users: this form of error recovery is strictly more limited than that provided by `yacc`. During a parse error condition, `yacc` attempts to discard states and tokens in order to get back into a state where parsing may continue;

Happy doesn't do this. The reason is that normal `yacc` error recovery is notoriously hard to describe, and the semantics depend heavily on the workings of a shift-reduce parser. Furthermore, different implementations of `yacc` appear to implement error recovery differently. Happy's limited error recovery on the other hand is well-defined, as is just sufficient to implement the Haskell layout rule (which is why it was added in the first place).

2.7 Generating Multiple Parsers From a Single Grammar

It is often useful to use a single grammar to describe multiple parsers, where each parser has a different top-level non-terminal, but parts of the grammar are shared between parsers. A classic example of this is an interpreter, which needs to be able to parse both entire files and single expressions: the expression grammar is likely to be identical for the two parsers, so we would like to use a single grammar but have two entry points.

Happy lets you do this by allowing multiple `%name` directives in the grammar file. The `%name` directive takes an optional second parameter specifying the top-level non-terminal for this parser, so we may specify multiple parsers like so:

```
%name parse1 non-terminal1
%name parse2 non-terminal2
```

Happy will generate from this a module which defines two functions `parse1` and `parse2`, which parse the grammars given by `non-terminal1` and `non-terminal2` respectively. Each parsing function will of course have a different type, depending on the type of the appropriate non-terminal.

CHAPTER

THREE

GENERALIZED LR PARSING

This chapter explains how to use the GLR parsing extension, which allows Happy to parse ambiguous grammars and produce useful results. This extension is triggered with the `--glr` flag, which causes Happy to use a different driver for the LALR(1) parsing tables. The result of parsing is a structure which encodes compactly *all* of the possible parses. There are two options for how semantic information is combined with the structural information.

This extension was developed by Paul Callaghan and Ben Medlock (University of Durham). It is based on the structural parser implemented in Medlock's undergraduate project, but significantly extended and improved by Callaghan. Bug reports, comments, questions etc should be sent to P.C.Callaghan@durham.ac.uk. Further information can be found on Callaghan's GLR parser page.

3.1 Introduction

Here's an ambiguous grammar. It has no information about the associativity of +, so for example, `1+2+3` can be parsed as `(1+(2+3))` or `((1+2)+3)`. In conventional mode, Happy, would complain about a shift/reduce conflict, although it would generate a parser which always shifts in such a conflict, and hence would produce *only* the first alternative above.

```
E -> E + E
E -> i       -- any integer
```

GLR parsing will accept this grammar without complaint, and produce a result which encodes *both* alternatives simultaneously. Now consider the more interesting example of `1+2+3+4`, which has five distinct parses — try to list them! You will see that some of the subtrees are identical. A further property of the GLR output is that such sub-results are shared, hence efficiently represented: there is no combinatorial explosion. Below is the simplified output of the GLR parser for this example.

```
Root (0,7,G_E)
(0,1,G_E)     => [[(0,1,Tok '1'))]]
(0,3,G_E)     => [[(0,1,G_E),(1,2,Tok '+'),(2,3,G_E)]]
(0,5,G_E)     => [[(0,1,G_E),(1,2,Tok '+'),(2,5,G_E)]
                  ,[(0,3,G_E),(3,4,Tok '+'),(4,5,G_E)]]
(0,7,G_E)     => [[(0,3,G_E),(3,4,Tok '+'),(4,7,G_E)]
                  ,[(0,1,G_E),(1,2,Tok '+'),(2,7,G_E)]
                  ,[(0,5,G_E),(5,6,Tok '+'),(6,7,G_E)]}]
(2,3,G_E)     => [[(2,3,Tok '2'))]}]
(2,5,G_E)     => [[(2,3,G_E),(3,4,Tok '+'),(4,5,G_E)]}]
(2,7,G_E)     => [[(2,3,G_E),(3,4,Tok '+'),(4,7,G_E)]}
                  ,[(2,5,G_E),(5,6,Tok '+'),(6,7,G_E)]}]
(4,5,G_E)     => [[(4,5,Tok '3'))]}]
(4,7,G_E)     => [[(4,5,G_E),(5,6,Tok '+'),(6,7,G_E)]}]
(6,7,G_E)     => [[(6,7,Tok '4'))]}]
```

This is a directed, acyclic and-or graph. The node "names" are of form `(a,b,c)` where `a` and `b` are the start and end points (as positions in the input string) and `c` is a category (or name of grammar rule). For example `(2,7,G_E)` spans positions 2 to 7 and contains analyses which match the `E` grammar rule. Such analyses are given as a list of alternatives (disjunctions), each corresponding to some use of a production of that category, which in turn are a conjunction of sub-analyses, each represented as a node in the graph or an instance of a token.

Hence `(2,7,G_E)` contains two alternatives, one which has `(2,3,G_E)` as its first child and the other with `(2,5,G_E)` as its first child, respectively corresponding to sub-analyses `(2+(3+4))` and `((2+3)+4)`. Both alternatives have the token + as their second child, but note that they are difference occurrences of + in the input! We strongly recommend looking at such results in graphical form to understand these points. If you build the `expr-eval` example in the directory `examples/glr` (N.B. you need to use GHC for this, unless you know how to use the `-F` flag for Hugs), running the example will produce a file which can be viewed with the *daVinci* graph visualization tool. (See http://www.informatik.uni-bremen.de/~davinci/ for more information. Educational use licenses are currently available without charge.)

The GLR extension also allows semantic information to be attached to productions, as in conventional Happy, although there are further issues to consider. Two modes are provided, one for simple applications and one for more complex use. See *Including semantic results*. The extension is also integrated with Happy's token handling, e.g. extraction of information from tokens.

One key feature of this implementation in Haskell is that its main result is a *graph*. Other implementations effectively produce a list of trees, but this limits practical use to small examples. For large and interesting applications, some of which are discussed in *Some Applications of GLR parsing*, a graph is essential due to the large number of possibilities and the need to analyse the structure of the ambiguity. Converting the graph to trees could produce huge numbers of results and will lose information about sharing etc.

One final comment. You may have learnt through using yacc-style tools that ambiguous grammars are to be avoided, and that ambiguity is something that appears only in Natural Language processing. This is definitely not true. Many interesting grammars are ambiguous, and with GLR tools they can be used effectively. We hope you enjoy exploring this fascinating area!

3.2 Basic use of a Happy-generated GLR parser

This section explains how to generate and to use a GLR parser to produce structural results. Please check the examples for further information. Discussion of semantic issues comes later; see *Including semantic results*.

3.2.1 Overview

The process of generating a GLR parser is broadly the same as for standard Happy. You write a grammar specification, run Happy on this to generate some Haskell code, then compile and link this into your program.

An alternative to using Happy directly is to use the BNF Converter tool by Markus Forsberg, Peter Gammie, Michael Pellauer and Aarne Ranta. This tool creates an abstract syntax, grammar, pretty-printer and other useful items from a single grammar formalism, thus it saves a lot of work and improves maintainability. The current output of BNFC can be used with GLR mode now with just a few small changes, but from January 2005 we expect to have a fully-compatible version of BNFC.

Most of the features of Happy still work, but note the important points below.

module header
: The GLR parser is generated in TWO files, one for data and one for the driver. This is because the driver code needs to be optimized, but for large parsers with lots of data, optimizing the data tables too causes compilation to be too slow. Given a file `Foo.y`, the file `FooData.hs`, containing the data module, is generated with basic type information, the parser tables, and the header and tail code that was included in the parser specification. Note that Happy can automatically generate the necessary module declaration statements, if you do not choose to provide one in the grammar file. But, if you do choose to provide the module declaration statement, then the

name of the module will be parsed and used as the name of the driver module. The parsed name will also be used to form the name of the data module, but with the string `Data` appended to it. The driver module, which is to be found in the file `Foo.hs`, will not contain any other user-supplied text besides the module name. Do not bother to supply any export declarations in your module declaration statement: they will be ignored and dropped, in favor of the standard export declaration.

export of lexer
: You can declare a lexer (and error token) with the `%lexer` directive as normal, but the generated parser does NOT call this lexer automatically. The action of the directive is only to *export* the lexer function to the top level. This is because some applications need finer control of the lexing process.

precedence information
: This still works, but note the reasons. The precedence and associativity declarations are used in Happy's LR table creation to resolve certain conflicts. It does this by retaining the actions implied by the declarations and removing the ones which clash with these. The GLR parser back-end then produces code from these filtered tables, hence the rejected actions are never considered by the GLR parser.

  Hence, declaring precedence and associativity is still a good thing, since it avoids a certain amount of ambiguity that the user knows how to remove.

monad directive
: There is some support for monadic parsers. The "tree decoding" mode (see *Tree decoding*) can use the information given in the `%monad` declaration to monadify the decoding process. This is explained in more detail in *Monadic tree decoding*.

  Note: the generated parsers don't include Ashley Yakeley's monad context information yet. It is currently just ignored. If this is a problem, email and I'll make the changes required.

parser name directive
: This has no effect at present. It will probably remain this way: if you want to control names, you could use qualified import.

type information on non-terminals
: The generation of semantic code relies on type information given in the grammar specification. If you don't give an explicit signature, the type `()` is assumed. If you get type clashes mentioning `()` you may need to add type annotations. Similarly, if you don't supply code for the semantic rule portion, then the value `()` is used.

`error` symbol in grammars, and recovery
: No attempt to implement this yet. Any use of `error` in grammars is thus ignored, and parse errors will eventually mean a parse will fail.

the token type
: The type used for tokens *must* be in the `Ord` type class (and hence in `Eq`). It is also recommended that they are in the `Show` class too. The ordering is required for the implementation of ambiguity packing. It may be possible to relax this requirement, but it is probably simpler just to require instances of the type classes. Please tell us if this is a problem.

3.2.2 The main function

The driver file exports a function `doParse :: [[UserDefTok]] -> GLRResult`. If you are using several parsers, use qualified naming to distinguish them. `UserDefTok` is a synonym for the type declared with the `%tokentype` directive.

3.2.3 The input

The input to `doParse` is a list of *list of* token values. The outer level represents the sequence of input symbols, and the inner list represents ambiguity in the tokenisation of each input symbol. For example, the word "run" can be at least a noun or a verb, hence the inner list will contain at least two values. If your tokens are not ambiguous, you will need to convert each token to a singleton list before parsing.

3.2.4 The Parse Result

The parse result is expressed with the following types. A successful parse yields a forest (explained below) and a single root node for the forest. A parse may fail for one of two reasons: running out of input or a (global) parse error. A global parse error means that it was not possible to continue parsing *any* of the live alternatives; this is different from a local error, which simply means that the current alternative dies and we try some other alternative. In both error cases, the forest at failure point is returned, since it may contain useful information. Unconsumed tokens are returned when there is a global parse error.

```
type ForestId = (Int,Int,GSymbol)
data GSymbol  = <... automatically generated ...>
type Forest   = FiniteMap ForestId [Branch]
type RootNode = ForestId
type Tokens   = [[(Int, GSymbol)]]
data Branch   = Branch {b_sem :: GSem, b_nodes :: [ForestId]}
data GSem     = <... automatically generated ...>

data GLRResult
  = ParseOK     RootNode Forest    -- forest with root
  | ParseError  Tokens   Forest    -- partial forest with bad input
  | ParseEOF             Forest    -- partial forest (missing input)
```

Conceptually, the parse forest is a directed, acyclic and-or graph. It is represented by a mapping of `ForestId`s to lists of possible analyses. The `FiniteMap` type is used to provide efficient and convenient access. The `ForestId` type identifies nodes in the graph, named by the range of input they span and the category of analysis they license. `GSymbol` is generated automatically as a union of the names of grammar rules (prefixed by `G_` to avoid name clashes) and of tokens and an EOF symbol. Tokens are wrapped in the constructor `HappyTok :: UserDefTok -> GSymbol`.

The `Branch` type represents a match for some right-hand side of a production, containing semantic information (see below) and a list of sub-analyses. Each of these is a node in the graph. Note that tokens are represented as childless nodes that span one input position. Empty productions will appear as childless nodes that start and end at the same position.

3.2.5 Compiling the parser

Happy will generate two files, and these should be compiled as normal Haskell files. If speed is an issue, then you should use the `-O` flags etc with the driver code, and if feasible, with the parser tables too.

You can also use the `--ghc` flag to trigger certain GHC-specific optimizations. At present, this just causes use of unboxed types in the tables and in some key code. Using this flag causes relevant GHC option pragmas to be inserted into the generated code, so you shouldn't have to use any strange flags (unless you want to...).

3.3 Including semantic results

This section discusses the options for including semantic information in grammars.

3.3.1 Forms of semantics

Semantic information may be attached to productions in the conventional way, but when more than one analysis is possible, the use of the semantic information must change. Two schemes have been implemented, which we call *tree decoding* and *label decoding*. The former is for simple applications, where there is not much ambiguity and hence where the effective unpacking of the parse forest isn't a factor. This mode is quite similar to the standard mode in Happy. The latter is for serious applications, where sharing is important and where processing of the forest (eg filtering) is needed. Here, the emphasis is about providing rich labels in nodes of the the parse forest, to support such processing.

The default mode is labelling. If you want the tree decode mode, use the `--decode` flag.

3.3.2 Tree decoding

Tree decoding corresponds to unpacking the parse forest to individual trees and collecting the list of semantic results computed from each of these. It is a mode intended for simple applications, where there is limited ambiguity. You may access semantic results from components of a reduction using the dollar variables. As a working example, the following is taken from the `expr-tree` grammar in the examples. Note that the type signature is required, else the types in use can't be determined by the parser generator.

```
E :: {Int} -- type signature needed
  : E '+' E  { $1 + $3 }
  | E '*' E  { $1 * $3 }
  | i        { $1 }
```

This mode works by converting each of the semantic rules into functions (abstracted over the dollar variables mentioned), and labelling each `Branch` created from a reduction of that rule with the function value. This amounts to *delaying* the action of the rule, since we must wait until we know the results of all of the sub-analyses before computing any of the results. (Certain cases of packing can add new analyses at a later stage.)

At the end of parsing, the functions are applied across relevant sub-analyses via a recursive descent. The main interface to this is via the class and entry function below. Typically, `decode` should be called on the root of the forest, also supplying a function which maps node names to their list of analyses (typically a partial application of lookup in the forest value). The result is a list of semantic values. Note that the context of the call to `decode` should (eventually) supply a concrete type to allow selection of appropriate instance. I.e., you have to indicate in some way what type the semantic result should have. `Decode_Result a` is a synonym generated by Happy: for non-monadic semantics, it is equivalent to `a`; when monads are in use, it becomes the declared monad type. See the full `expr-eval` example for more information.

```
class TreeDecode a where
        decode_b :: (ForestId -> [Branch]) -> Branch -> [Decode_Result a]
decode :: TreeDecode a => (ForestId -> [Branch]) -> ForestId -> [Decode_Result a]
```

The GLR parser generator identifies the types involved in each semantic rule, hence the types of the functions, then creates a union containing distinct types. Values of this union are stored in the branches. (The union is actually a bit more complex: it must also distinguish patterns of dollar-variable usage, eg a function `\x y -> x + y` could be applied to the first and second constituents, or to the first and third.) The parser generator also creates instances of the `TreeDecode` class, which unpacks the semantic function and applies it across the decodings of the possible combinations of children. Effectively, it does a Cartesian product operation across the lists of semantic results from each of the children. Eg `[1,2] "+" [3,4]` produces `[4,5,5,6]`. Information is extracted from token values using

the patterns supplied by the user when declaring tokens and their Haskell representation, so the dollar-dollar convention works also.

The decoding process could be made more efficient by using memoisation techniques, but this hasn't been implemented since we believe the other (label) decoding mode is more useful. (If someone sends in a patch, we may include it in a future release — but this might be tricky, e.g. require higher-order polymorphism? Plus, are there other ways of using this form of semantic function?)

3.3.3 Label decoding

The labelling mode aims to label branches in the forest with information that supports subsequent processing, for example the filtering and prioritisation of analyses prior to extraction of favoured solutions. As above, code fragments are given in braces and can contain dollar-variables. But these variables are expanded to node names in the graph, with the intention of easing navigation. The following grammar is from the `expr-tree` example.

```
E :: {Tree ForestId Int}
  : E '+' E      { Plus  $1 $3 }
  | E '*' E      { Times $1 $3 }
  | i            { Const $1 }
```

Here, the semantic values provide more meaningful labels than the plain structural information. In particular, only the interesting parts of the branch are represented, and the programmer can clearly select or label the useful constituents if required. There is no need to remember that it is the first and third child in the branch which we need to extract, because the label only contains those values (the `noise' has been dropped). Consider also the difference between concrete and abstract syntax. The labels are oriented towards abstract syntax. Tokens are handled slightly differently here: when they appear as children in a reduction, their informational content can be extracted directly, hence the `Const` value above will be built with the `Int` value from the token, not some `ForestId`.

Note the useful technique of making the label types polymorphic in the position used for forest indices. This allows replacement at a later stage with more appropriate values, e.g. inserting lists of actual subtrees from the final decoding.

Use of these labels is supported by a type class `LabelDecode`, which unpacks values of the automatically-generated union type `GSem` to the original type(s). The parser generator will create appropriate instances of this class, based on the type information in the grammar file. (Note that omitting type information leads to a default of `()`.) Observe that use of the labels is often like traversing an abstract syntax, and the structure of the abstract syntax type usually constrains the types of constituents; so once the overall type is fixed (e.g. with a type cast or signature) then there are no problems with resolution of class instances.

```
class LabelDecode a where
        unpack :: GSem -> a
```

Internally, the semantic values are packed in a union type as before, but there is no direct abstraction step. Instead, the `ForestId` values (from the dollar-variables) are bound when the corresponding branch is created from the list of constituent nodes. At this stage, token information is also extracted, using the patterns supplied by the user when declaring the tokens.

3.3.4 Monadic tree decoding

You can use the `%monad` directive in the tree-decode mode. Essentially, the decoding process now creates a list of monadic values, using the monad type declared in the directive. The default handling of the semantic functions is to apply the relevant `return` function to the value being returned. You can over-ride this using the `{% ... }` convention. The declared `(>>=)` function is used to assemble the computations.

Note that no attempt is made to share the results of monadic computations from sub-trees. (You could possibly do this by supplying a memoising lookup function for the decoding process.) Hence, the usual behaviour is that decoding produces whole monadic computations, each part of which is computed afresh (in depth-first order) when the whole is computed. Hence you should take care to initialise any relevant state before computing the results from multiple solutions.

This facility is experimental, and we welcome comments or observations on the approach taken! An example is provided (`examples/glr/expr-monad`). It is the standard example of arithmetic expressions, except that the `IO` monad is used, and a user exception is thrown when the second argument to addition is an odd number. Running this example will show a zero (from the exception handler) instead of the expected number amongst the results from the other parses.

Further information ——————-

Other useful information...

The GLR examples ~~~~~~~~~~~~~~~~~

The directory `examples/glr` contains several examples from the small to the large. Please consult these or use them as a base for your experiments.

3.3.5 Viewing forests as graphs

If you run the examples with GHC, each run will produce a file `out.daVinci`. This is a graph in the format expected by the *daVinci* graph visualization tool. (See http://www.informatik.uni-bremen.de/~davinci/ for more information. Educational use licenses are currently available without charge.)

We highly recommend looking at graphs of parse results — it really helps to understand the results. The graphs files are created with Sven Panne's library for communicating with *daVinci*, supplemented with some extensions due to Callaghan. Copies of this code are included in the examples directory, for convenience. If you are trying to view large and complex graphs, contact Paul Callaghan (there are tools and techniques to make the graphs more manageable).

3.3.6 Some Applications of GLR parsing

GLR parsing (and related techniques) aren't just for badly written grammars or for things like natural language (NL) where ambiguity is inescapable. There are applications where ambiguity can represent possible alternatives in pattern-matching tasks, and the flexibility of these parsing techniques and the resulting graphs support deep analyses. Below, we briefly discuss some examples, a mixture from our recent work and from the literature.

Gene sequence analysis
: Combinations of structures within gene sequences can be expressed as a grammar, for example a "start" combination followed by a "promoter" combination then the gene proper. A recent undergraduate project has used this GLR implementation to detect candiate matches in data, and then to filter these matches with a mixture of local and global information.

Rhythmic structure in poetry
: Rhythmic patterns in (English) poetry obey certain rules, and in more modern poetry can break rules in particular ways to achieve certain effects. The standard rhythmic patterns (e.g. iambic pentameter) can be encoded as a grammar, and deviations from the patterns also encoded as rules. The neutral reading can be parsed with this grammar, to give a forest of alternative matches. The forest can be analysed to give a preferred reading,

and to highlight certain technical features of the poetry. An undergraduate project in Durham has used this implementation for this purpose, with promising results.

Compilers — instruction selection
Recent work has phrased the translation problem in compilers from intermediate representation to an instruction set for a given processor as a matching problem. Different constructs at the intermediate level can map to several combinations of machine instructions. This knowledge can be expressed as a grammar, and instances of the problem solved by parsing. The parse forest represents competing solutions, and allows selection of optimum solutions according to various measures.

Robust parsing of ill-formed input
The extra flexibility of GLR parsing can simplify parsing of formal languages where a degree of `informality' is allowed. For example, Html parsing. Modern browsers contain complex parsers which are designed to try to extract useful information from Html text which doesn't follow the rules precisely, eg missing start tags or missing end tags. Html with missing tags can be written as an ambiguous grammar, and it should be a simple matter to extract a usable interpretation from a forest of parses. Notice the technique: we widen the scope of the grammar, parse with GLR, then extract a reasonable solution. This is arguably simpler than pushing an LR(1) or LL(1) parser past its limits, and also more maintainable.

Natural Language Processing
Ambiguity is inescapable in the syntax of most human languages. In realistic systems, parse forests are useful to encode competing analyses in an efficient way, and they also provide a framework for further analysis and disambiguation. Note that ambiguity can have many forms, from simple phrase attachment uncertainty to more subtle forms involving mixtures of word senses. If some degree of ungrammaticality is to be tolerated in a system, which can be done by extending the grammar with productions incorporating common forms of infelicity, the degree of ambiguity increases further. For systems used on arbitrary text, such as on newspapers, it is not uncommon that many sentences permit several hundred or more analyses. With such grammars, parse forest techniques are essential. Many recent NLP systems use such techniques, including the Durham's earlier LOLITA system - which was mostly written in Haskell.

3.3.7 Technical details

The original implementation was developed by Ben Medlock, as his undergraduate final year project, using ideas from Peter Ljunglöf's Licentiate thesis (see https://gup.ub.gu.se/publication/10783, and we recommend the thesis for its clear analysis of parsing algorithms). Ljunglöf's version produces lists of parse trees, but Medlock adapted this to produce an explicit graph containing parse structure information. He also incorporated the code into Happy.

After Medlock's graduation, Callaghan extended the code to incorporate semantic information, and made several improvements to the original code, such as improved local packing and support for hidden left recursion. The performance of the code was significantly improved, after changes of representation (eg to a chart-style data structure) and technique. Medlock's code was also used in several student projects, including analysis of gene sequences (Fischer) and analysis of rhythmic patterns in poetry (Henderson).

The current code implements the standard GLR algorithm extended to handle hidden left recursion. Such recursion, as in the grammar below from [Rekers1992], causes the standard algorithm to loop because the empty reduction `A ->` is always possible and the LR parser will not change state. Alternatively, there is a problem because an unknown (at the start of parsing) number of `A` items are required, to match the number of `i` tokens in the input.

```
S -> A Q i | +
A ->
```

The solution to this is not surprising. Problematic recursions are detected as zero-span reductions in a state which has a `goto` table entry looping to itself. A special symbol is pushed to the stack on the first such reduction, and such reductions are done at most once for any token alternative for any input position. When popping from the stack, if the last token being popped is such a special symbol, then two stack tails are returned: one corresponding to a conventional pop (which removes the symbol) and the other to a duplication of the special symbol (the stack is not changed, but

a copy of the symbol is returned). This allows sufficient copies of the empty symbol to appear on some stack, hence allowing the parse to complete.

The forest is held in a chart-style data structure, and this supports local ambiguity packing (chart parsing is discussed in Ljunglöf's thesis, among other places). A limited amount of packing of live stacks is also done, to avoid some repetition of work.

3.3.8 The `--filter` option

You might have noticed this GLR-related option. It is an experimental feature intended to restrict the amount of structure retained in the forest by discarding everything not required for the semantic results. It may or it may not work, and may be fixed in a future release.

3.3.9 Limitations and future work

The parser supports hidden left recursion, but makes no attempt to handle cyclic grammars that have rules which do not consume any input. If you have a grammar like this, for example with rules like `S -> S` or `S -> A S | x; A -> empty`, the implementation will loop until you run out of stack - but if it will happen, it often happens quite quickly!

The code has been used and tested frequently over the past few years, including being used in several undergraduate projects. It should be fairly stable, but as usual, can't be guaranteed bug-free. One day I will write it in Epigram!

If you have suggestions for improvements, or requests for features, please contact Paul Callaghan. There are some changes I am considering, and some views and/or encouragement from users will be much appreciated. Further information can be found on Callaghan's GLR parser page.

3.3.10 Thanks and acknowledgements

Many thanks to the people who have used and tested this software in its various forms, including Julia Fischer, James Henderson, and Aarne Ranta.

CHAPTER

FOUR

ATTRIBUTE GRAMMARS

4.1 Introduction

Attribute grammars are a formalism for expressing syntax directed translation of a context-free grammar. An introduction to attribute grammars may be found here. There is also an article in the Monad Reader about attribute grammars and a different approach to attribute grammars using Haskell here.

The main practical difficulty that has prevented attribute grammars from gaining widespread use involves evaluating the attributes. Attribute grammars generate non-trivial data dependency graphs that are difficult to evaluate using mainstream languages and techniques. The solutions generally involve restricting the form of the grammars or using big hammers like topological sorts. However, a language which supports lazy evaluation, such as Haskell, has no problem forming complex data dependency graphs and evaluating them. The primary intellectual barrier to attribute grammar adoption seems to stem from the fact that most programmers have difficulty with the declarative nature of the specification. Haskell programmers, on the other hand, have already embraced a purely functional language. In short, the Haskell language and community seem like a perfect place to experiment with attribute grammars.

Embedding attribute grammars in Happy is easy because because Haskell supports three important features: higher order functions, labeled records, and lazy evaluation. Attributes are encoded as fields in a labeled record. The parse result of each non-terminal in the grammar is a function which takes a record of inherited attributes and returns a record of synthesized attributes. In each production, the attributes of various non-terminals are bound together using `let`. Finally, at the end of the parse, a distinguished attribute is evaluated to be the final result. Lazy evaluation takes care of evaluating each attribute in the correct order, resulting in an attribute grammar system that is capable of evaluating a fairly large class of attribute grammars.

Attribute grammars in Happy do not use any language extensions, so the parsers are Haskell 98 (assuming you don't use the GHC specific `-g` option). Currently, attribute grammars cannot be generated for GLR parsers. (It's not exactly clear how these features should interact...)

4.2 Attribute Grammars in Happy

4.2.1 Declaring Attributes

The presence of one or more `%attribute` directives indicates that a grammar is an attribute grammar. Attributes are calculated properties that are associated with the non-terminals in a parse tree. Each `%attribute` directive generates a field in the attributes record with the given name and type.

The first `%attribute` directive in a grammar defines the default attribute. The default attribute is distinguished in two ways: 1) if no attribute specifier is given on an attribute reference, the default attribute is assumed (see *Semantic Rules*) and 2) the value for the default attribute of the starting non-terminal becomes the return value of the parse.

Optionally, one may specify a type declaration for the attribute record using the `%attributetype` declaration. This allows you to define the type given to the attribute record and, more importantly, allows you to introduce type variables

that can be subsequently used in `%attribute` declarations. If the `%attributetype` directive is given without any `%attribute` declarations, then the `%attributetype` declaration has no effect.

For example, the following declarations:

```
%attributetype { MyAttributes a }
%attribute value { a }
%attribute num   { Int }
%attribute label { String }
```

would generate this attribute record declaration in the parser:

```
data MyAttributes a =
   HappyAttributes {
     value :: a,
     num :: Int,
     label :: String
   }
```

and `value` would be the default attribute.

4.2.2 Semantic Rules

In an ordinary Happy grammar, a production consists of a list of terminals and/or non-terminals followed by an uninterpreted code fragment enclosed in braces. With an attribute grammar, the format is very similar, but the braces enclose a set of semantic rules rather than uninterpreted Haskell code. Each semantic rule is either an attribute calculation or a conditional, and rules are separated by semicolons[3].

Both attribute calculations and conditionals may contain attribute references and/or terminal references. Just like regular Happy grammars, the tokens `$1` through `$<n>`, where `n` is the number of symbols in the production, refer to subtrees of the parse. If the referenced symbol is a terminal, then the value of the reference is just the value of the terminal, the same way as in a regular Happy grammar. If the referenced symbol is a non-terminal, then the reference may be followed by an attribute specifier, which is a dot followed by an attribute name. If the attribute specifier is omitted, then the default attribute is assumed (the default attribute is the first attribute appearing in an `%attribute` declaration). The special reference `$$` references the attributes of the current node in the parse tree; it behaves exactly like the numbered references. Additionally, the reference `$>` always references the rightmost symbol in the production.

An attribute calculation rule is of the form:

```
<attribute reference> = <Haskell expression>
```

A rule of this form defines the value of an attribute, possibly as a function of the attributes of `$$` (inherited attributes), the attributes of non-terminals in the production (synthesized attributes), or the values of terminals in the production. The value for an attribute can only be defined once for a particular production.

The following rule calculates the default attribute of the current production in terms of the first and second items of the production (a synthesized attribute):

```
$$ = $1 : $2
```

This rule calculates the length attribute of a non-terminal in terms of the length of the current non-terminal (an inherited attribute):

[3] Note that semantic rules must not rely on layout, because whitespace alignment is not guaranteed to be preserved

```
$1.length = $$.length + 1
```

Conditional rules allow the rejection of strings due to context-sensitive properties. All conditional rules have the form:

```
where <Haskell expression>
```

For non-monadic parsers, all conditional expressions must be of the same (monomorphic) type. At the end of the parse, the conditionals will be reduced using `seq`, which gives the grammar an opportunity to call `error` with an informative message. For monadic parsers, all conditional statements must have type `Monad m => m ()` where `m` is the monad in which the parser operates. All conditionals will be sequenced at the end of the parse, which allows the conditionals to call `fail` with an informative message.

The following conditional rule will cause the (non-monadic) parser to fail if the inherited length attribute is not 0.

```
where if $$.length == 0 then () else error "length not equal to 0"
```

This conditional is the monadic equivalent:

```
where unless ($$.length == 0) (fail "length not equal to 0")
```

4.3 Limits of Happy Attribute Grammars

If you are not careful, you can write an attribute grammar which fails to terminate. This generally happens when semantic rules are written which cause a circular dependency on the value of an attribute. Even if the value of the attribute is well-defined (that is, if a fixpoint calculation over attribute values will eventually converge to a unique solution), this attribute grammar system will not evaluate such grammars.

One practical way to overcome this limitation is to ensure that each attribute is always used in either a top-down (inherited) fashion or in a bottom-up (synthesized) fashion. If the calculations are sufficiently lazy, one can "tie the knot" by synthesizing a value in one attribute, and then assigning that value to another, inherited attribute at some point in the parse tree. This technique can be useful for common tasks like building symbol tables for a syntactic scope and making that table available to sub-nodes of the parse.

4.4 Example Attribute Grammars

The following two toy attribute grammars may prove instructive. The first is an attribute grammar for the classic context-sensitive grammar { a^n b^n c^n | n >= 0 }. It demonstrates the use of conditionals, inherited and synthesized attributes.

```
{
module ABCParser (parse) where
}

%tokentype { Char }

%token a { 'a' }
%token b { 'b' }
%token c { 'c' }
%token newline { '\n' }

%attributetype { Attrs a }
```

(continues on next page)

(continued from previous page)

```
%attribute value { a }
%attribute len   { Int }

%name parse abcstring

%%

abcstring
   : alist blist clist newline
        { $$ = $1 ++ $2 ++ $3
        ; $2.len = $1.len
        ; $3.len = $1.len
        }

alist
   : a alist
        { $$ = $1 : $2
        ; $$.len = $2.len + 1
        }
   |    { $$ = []; $$.len = 0 }

blist
   : b blist
        { $$ = $1 : $2
        ; $2.len = $$.len - 1
        }
   |    { $$ = []
        ; where failUnless ($$.len == 0) "blist wrong length"
        }

clist
   : c clist
        { $$ = $1 : $2
        ; $2.len = $$.len - 1
        }
   |    { $$ = []
        ; where failUnless ($$.len == 0) "clist wrong length"
        }

{
happyError = error "parse error"
failUnless b msg = if b then () else error msg
}
```

This grammar parses binary numbers and calculates their value. It demonstrates the use of inherited and synthesized attributes.

```
{
module BitsParser (parse) where
}

%tokentype { Char }
```

(continues on next page)

(continued from previous page)

```
%token minus { '-' }
%token plus  { '+' }
%token one   { '1' }
%token zero  { '0' }
%token newline { '\n' }

%attributetype { Attrs }
%attribute value { Integer }
%attribute pos   { Int }

%name parse start

%%

start
   : num newline { $$ = $1 }

num
   : bits        { $$ = $1       ; $1.pos = 0 }
   | plus bits   { $$ = $2       ; $2.pos = 0 }
   | minus bits  { $$ = negate $2; $2.pos = 0 }

bits
   : bit         { $$ = $1
                 ; $1.pos = $$.pos
                 }

   | bits bit    { $$ = $1 + $2
                 ; $1.pos = $$.pos + 1
                 ; $2.pos = $$.pos
                 }

bit
   : zero        { $$ = 0 }
   | one         { $$ = 2^($$.pos) }

{
happyError = error "parse error"
}
```

CHAPTER

FIVE

INVOKING HAPPY

An invocation of Happy has the following syntax:

```
$ happy [ options ] filename [ options ]
```

All the command line options are optional (!) and may occur either before or after the input file name. Options that take arguments may be given multiple times, and the last occurrence will be the value used.

There are two types of grammar files, `file.y` and `file.ly`, with the latter observing the reverse comment (or literate) convention (i.e. each code line must begin with the character >, lines which don't begin with > are treated as comments). The examples distributed with Happy are all of the `.ly` form.

The flags accepted by Happy are as follows:

`-o <file>; --outfile=<file>`
: Specifies the destination of the generated parser module. If omitted, the parser will be placed in <file>`.hs`, where <file> is the name of the input file with any extension removed.

`-i [<file>]; --info[=<file>]`

: Directs Happy to produce an info file containing detailed information about the grammar, parser states, parser actions, and conflicts. Info files are vital during the debugging of grammars. The filename argument is optional. (note that there's no space between `-i` and the filename in the short version), and if omitted the info file will be written to <file>`.info` (where <file> is the input file name with any extension removed).

`-p [<file>]; --pretty[=<file>]`

: Directs Happy to produce a file containing a pretty-printed form of the grammar, containing only the productions, without any semantic actions or type signatures. If no file name is provided, then the file name will be computed by replacing the extension of the input file with `.grammar`.

`-t <dir>; --template=<dir>`

: Instructs Happy to use this directory when looking for template files: these files contain the static code that Happy includes in every generated parser. You shouldn't need to use this option if Happy is properly configured for your computer.

`-m <name>; --magic-name=<name>`
: Happy prefixes all the symbols it uses internally with either `happy` or `Happy`. To use a different string, for example if the use of `happy` is conflicting with one of your own functions, specify the prefix using the `-m` option.

`-s; --strict`

: **Warning:** The `--strict` option is experimental and may cause unpredictable results.

This option causes the right hand side of each production (the semantic value) to be evaluated eagerly at the moment the production is reduced. If the lazy behaviour is not required, then using this option will improve performance and may reduce space leaks. Note that the parser as a whole is never lazy - the whole input will always be consumed before any input is produced, regardless of the setting of the `--strict` flag.

`-g`; `--ghc`

Instructs Happy to generate a parser that uses GHC-specific extensions to obtain faster code.

`-c`; `--coerce`

Use GHC's `unsafeCoerce#` extension to generate smaller faster parsers. Type-safety isn't compromised.

This option may only be used in conjunction with `-g`.

`-a`; `--arrays`

Instructs Happy to generate a parser using an array-based shift reduce parser. When used in conjunction with `-g`, the arrays will be encoded as strings, resulting in faster parsers. Without `-g`, standard Haskell arrays will be used.

`-d`; `--debug`

Generate a parser that will print debugging information to `stderr` at run-time, including all the shifts, reductions, state transitions and token inputs performed by the parser.

This option can only be used in conjunction with `-a`.

`-l`; `--glr`

Generate a GLR parser for ambiguous grammars.

`-k`; `--decode`

Generate simple decoding code for GLR result.

`-f`; `--filter`

Filter the GLR parse forest with respect to semantic usage.

`-?`; `--help`
Print usage information on standard output then exit successfully.

`-V`; `--version`
Print version information on standard output then exit successfully. Note that for legacy reasons `-v` is supported, too, but the use of it is deprecated. `-v` will be used for verbose mode when it is actually implemented.

CHAPTER

SIX

SYNTAX OF GRAMMAR FILES

The input to Happy is a text file containing the grammar of the language you want to parse, together with some annotations that help the parser generator make a legal Haskell module that can be included in your program. This section gives the exact syntax of grammar files.

The overall format of the grammar file is given below:

```
<optional module header>
<directives>
%%
<grammar>
<optional module trailer>
```

If the name of the grammar file ends in `.ly`, then it is assumed to be a literate script. All lines except those beginning with a > will be ignored, and the > will be stripped from the beginning of all the code lines. There must be a blank line between each code section (lines beginning with >) and comment section. Grammars not using the literate notation must be in a file with the `.y` suffix.

6.1 Lexical Rules

Identifiers in Happy grammar files must take the following form (using the BNF syntax from the Haskell Report):

```
id      ::= alpha { idchar }
          | ' { any{^'} | \' } '
          | " { any{^"} | \" } "

alpha   ::= A | B | ... | Z
          | a | b | ... | z

idchar  ::= alpha
          | 0 | 1 | ... | 9
          | _
```

6.2 Module Header

This section is optional, but if included takes the following form:

```
{
<Haskell module header>
}
```

The Haskell module header contains the module name, exports, and imports. No other code is allowed in the header—this is because Happy may need to include its own `import` statements directly after the user defined header.

6.3 Directives

This section contains a number of lines of the form:

```
%<directive name> <argument> ...
```

The statements here are all annotations to help Happy generate the Haskell code for the grammar. Some of them are optional, and some of them are required.

6.3.1 Token Type

```
%tokentype   { <valid Haskell type> }
```

(mandatory) The `%tokentype` directive gives the type of the tokens passed from the lexical analyser to the parser (in order that Happy can supply types for functions and data in the generated parser).

6.3.2 Tokens

```
%token <name> { <Haskell pattern> }
       <name> { <Haskell pattern> }
       ...
```

(mandatory) The `%token` directive is used to tell Happy about all the terminal symbols used in the grammar. Each terminal has a name, by which it is referred to in the grammar itself, and a Haskell representation enclosed in braces. Each of the patterns must be of the same type, given by the `%tokentype` directive.

The name of each terminal follows the lexical rules for Happy identifiers given above. There are no lexical differences between terminals and non-terminals in the grammar, so it is recommended that you stick to a convention; for example using upper case letters for terminals and lower case for non-terminals, or vice-versa.

Happy will give you a warning if you try to use the same identifier both as a non-terminal and a terminal, or introduce an identifier which is declared as neither.

To save writing lots of projection functions that map tokens to their components, you can include `$$` in your Haskell pattern. For example:

```
%token INT { TokenInt $$ }
       ...
```

This makes the semantic value of `INT` refer to the first argument of `TokenInt` rather than the whole token, eliminating the need for any projection function.

6.3.3 Parser Name

```
%name <Haskell identifier> [ <non-terminal> ]
...
```

(optional) The `%name` directive is followed by a valid Haskell identifier, and gives the name of the top-level parsing function in the generated parser. This is the only function that needs to be exported from a parser module.

If the `%name` directive is omitted, it defaults to `happyParse`.

The `%name` directive takes an optional second parameter which specifies the top-level non-terminal which is to be parsed. If this parameter is omitted, it defaults to the first non-terminal defined in the grammar.

Multiple `%name` directives may be given, specifying multiple parser entry points for this grammar (see *Generating Multiple Parsers From a Single Grammar*). When multiple `%name` directives are given, they must all specify explicit non-terminals.

6.3.4 Partial Parsers

```
%partial <Haskell identifier> [ <non-terminal> ]
...
```

The `%partial` directive can be used instead of `%name`. It indicates that the generated parser should be able to parse an initial portion of the input. In contrast, a parser specified with `%name` will only parse the entire input.

A parser specified with `%partial` will stop parsing and return a result as soon as there exists a complete parse, and no more of the input can be parsed. It does this by accepting the parse if it is followed by the `error` token, rather than insisting that the parse is followed by the end of the token stream (or the `eof` token in the case of a `%lexer` parser).

6.3.5 Monad Directive

```
%monad { <type> } { <then> } { <return> }
```

(optional) The `%monad` directive takes three arguments: the type constructor of the monad, the `then` (or `bind`) operation, and the `return` (or `unit`) operation. The type constructor can be any type with kind `* -> *`.

Monad declarations are described in more detail in *Monadic Parsers*.

6.3.6 Lexical Analyser

```
%lexer { <lexer> } { <eof> }
```

(optional) The `%lexer` directive takes two arguments: `<lexer>` is the name of the lexical analyser function, and `<eof>` is a token that is to be treated as the end of file.

Lexer declarations are described in more detail in *Threaded Lexers*.

6.3.7 Precedence declarations

```
%left     <name> ...
%right    <name> ...
%nonassoc <name> ...
```

These declarations are used to specify the precedences and associativity of tokens. The precedence assigned by a `%left`, `%right` or `%nonassoc` declaration is defined to be higher than the precedence assigned by all declarations earlier in the file, and lower than the precedence assigned by all declarations later in the file.

The associativity of a token relative to tokens in the same `%left`, `%right`, or `%nonassoc` declaration is to the left, to the right, or non-associative respectively.

Precedence declarations are described in more detail in *Using Precedences*.

6.3.8 Expect declarations

```
%expect <number>
```

(optional) More often than not the grammar you write will have conflicts. These conflicts generate warnings. But when you have checked the warnings and made sure that Happy handles them correctly these warnings are just annoying. The `%expect` directive gives a way of avoiding them. Declaring `%expect n` is a way of telling Happy "There are exactly <n> shift/reduce conflicts and zero reduce/reduce conflicts in this grammar. I promise I have checked them and they are resolved correctly". When processing the grammar, Happy will check the actual number of conflicts against the `%expect` declaration if any, and if there is a discrepancy then an error will be reported.

Happy's `%expect` directive works exactly like that of yacc.

6.3.9 Error declaration

```
%error { <identifier> }
```

Specifies the function to be called in the event of a parse error. The type of `<identifier>` varies depending on the presence of `%lexer` (see *Summary*) and `%errorhandlertype` (see the following).

6.3.10 Additional error information

```
%errorhandlertype (explist | default)
```

(optional) The expected type of the user-supplied error handling can be applied with additional information. By default, no information is added, for compatibility with previous versions. However, if `explist` is provided with this directive, then the first application will be of type `[String]`, providing a description of possible tokens that would not have failed the parser in place of the token that has caused the error.

6.3.11 Attribute Type Declaration

```
%attributetype { <valid Haskell type declaration> }
```

directive (optional) This directive allows you to declare the type of the attributes record when defining an attribute grammar. If this declaration is not given, Happy will choose a default. This declaration may only appear once in a grammar.

Attribute grammars are explained in *Attribute Grammars*.

6.3.12 Attribute declaration

```
%attribute <Haskell identifier> { <valid Haskell type> }
```

The presence of one or more of these directives declares that the grammar is an attribute grammar. The first attribute listed becomes the default attribute. Each `%attribute` directive generates a field in the attributes record with the given label and type. If there is an `%attributetype` declaration in the grammar which introduces type variables, then the type of an attribute may mention any such type variables.

Attribute grammars are explained in *Attribute Grammars*.

6.4 Grammar

The grammar section comes after the directives, separated from them by a double-percent (`%%`) symbol. This section contains a number of *productions*, each of which defines a single non-terminal. Each production has the following syntax:

```
<non-terminal> [ :: { <type> } ]
        :  <id> ... {[%] <expression> }
      [ |  <id> ... {[%] <expression> }
        ... ]
```

The first line gives the non-terminal to be defined by the production and optionally its type (type signatures for productions are discussed in *Type Signatures*).

Each production has at least one, and possibly many right-hand sides. Each right-hand side consists of zero or more symbols (terminals or non-terminals) and a Haskell expression enclosed in braces.

The expression represents the semantic value of the non-terminal, and may refer to the semantic values of the symbols in the right-hand side using the meta-variables `$1 ... $n`. It is an error to refer to `$i` when `i` is larger than the number of symbols on the right hand side of the current rule. The symbol `$` may be inserted literally in the Haskell expression using the sequence `\$` (this isn't necessary inside a string or character literal).

Additionally, the sequence `$>` can be used to represent the value of the rightmost symbol.

A semantic value of the form `{% ... }` is a *monadic action*, and is only valid when the grammar file contains a `%monad` directive (*Monad Directive*). Monadic actions are discussed in *Monadic Parsers*.

Remember that all the expressions for a production must have the same type.

6.4.1 Parameterized Productions

Starting from version 1.17.1, Happy supports *parameterized productions* which provide a convenient notation for capturing recurring patterns in context free grammars. This gives the benefits of something similar to parsing combinators in the context of Happy grammars.

This functionality is best illustrated with an example:

```
opt(p)          : p                 { Just $1 }
                |                   { Nothing }

rev_list1(p)    : p                 { [$1] }
                | rev_list1(p) p    { $2 : $1 }
```

The first production, `opt`, is used for optional components of a grammar. It is just like `p?` in regular expressions or EBNF. The second production, `rev_list1`, is for parsing a list of 1 or more occurrences of `p`. Parameterized productions are just like ordinary productions, except that they have parameter in parenthesis after the production name. Multiple parameters should be separated by commas:

```
fst(p,q)        : p q               { $1 }
snd(p,q)        : p q               { $2 }
both(p,q)       : p q               { ($1,$2) }
```

To use a parameterized production, we have to pass values for the parameters, as if we are calling a function. The parameters can be either terminals, non-terminals, or other instantiations of parameterized productions. Here are some examples:

```
list1(p)        : rev_list1(p)      { reverse $1 }
list(p)         : list1(p)          { $1 }
                |                   { [] }
```

The first production uses `rev_list` to define a production that behaves like `p+`, returning a list of elements in the same order as they occurred in the input. The second one, `list` is like `p*`.

Parameterized productions are implemented as a preprocessing pass in Happy: each instantiation of a production turns into a separate non-terminal, but are careful to avoid generating the same rule multiple times, as this would lead to an ambiguous grammar. Consider, for example, the following parameterized rule:

```
sep1(p,q)       : p list(snd(q,p))  { $1 : $2 }
```

The rules that would be generated for `sep1(EXPR,SEP)`

```
sep1(EXPR,SEP)
  : EXPR list(snd(SEP,EXPR))                { $1 : $2 }

list(snd(SEP,EXPR))
  : list1(snd(SEP,EXPR))                    { $1 }
  |                                         { [] }

list1(snd(SEP,EXPR))
  : rev_list1(snd(SEP,EXPR))                { reverse $1 }

rev_list1(snd(SEP,EXPR))
  : snd(SEP,EXPR))                          { [$1] }
  | rev_list1(snd(SEP,EXPR)) snd(SEP,EXPR)  { $2 : $1 }
```

(continues on next page)

(continued from previous page)

```
snd(SEP,EXPR)
  : SEP EXPR                                { $2 }
```

Note that this is just a normal grammar, with slightly strange names for the non-terminals.

A drawback of the current implementation is that it does not support type signatures for the parameterized productions, that depend on the types of the parameters. We plan to implement that in the future — the current workaround is to omit the type signatures for such rules.

6.5 Module Trailer

The module trailer is optional, comes right at the end of the grammar file, and takes the same form as the module header:

```
{
<Haskell code>
}
```

This section is used for placing auxiliary definitions that need to be in the same module as the parser. In small parsers, it often contains a hand-written lexical analyser too. There is no restriction on what can be placed in the module trailer, and any code in there is copied verbatim into the generated parser file.

CHAPTER

SEVEN

INFO FILES

Happy info files, generated using the `-i` flag, are your most important tool for debugging errors in your grammar. Although they can be quite verbose, the general concept behind them is quite simple.

An info file contains the following information:

1. A summary of all shift/reduce and reduce/reduce conflicts in the grammar.
2. Under section `Grammar`, a summary of all the rules in the grammar. These rules correspond directly to your input file, absent the actual Haskell code that is to be run for each rules. A rule is written in the form `<non-terminal> -> <id> ...`
3. Under section `Terminals`, a summary of all the terminal tokens you may run against, as well as a the Haskell pattern which matches against them. This corresponds directly to the contents of your `%token` directive (*Tokens*).
4. Under section `Non-terminals`, a summary of which rules apply to which productions. This is generally redundant with the `Grammar` section.
5. The primary section `States`, which describes the state-machine Happy built for your grammar, and all of the transitions for each state.
6. Finally, some statistics `Grammar Totals` at the end of the file.

In general, you will be most interested in the `States` section, as it will give you information, in particular, about any conflicts your grammar may have.

7.1 States

Although Happy does its best to insulate you from the vagaries of parser generation, it's important to know a little about how shift-reduce parsers work in order to be able to interpret the entries in the `States` section.

In general, a shift-reduce parser operates by maintaining parse stack, which tokens and productions are shifted onto or reduced off of. The parser maintains a state machine, which accepts a token, performs some shift or reduce, and transitions to a new state for the next token. Importantly, these states represent *multiple* possible productions, because in general the parser does not know what the actual production for the tokens it's parsing is going to be. There's no direct correspondence between the state-machine and the input grammar; this is something you have to reverse engineer.

With this knowledge in mind, we can look at two example states from the example grammar from *Using*:

```
State 5

        Exp1 -> Term .                                  (rule 5)
        Term -> Term . '*' Factor                       (rule 6)
        Term -> Term . '/' Factor                       (rule 7)
```

(continues on next page)

(continued from previous page)

```
        in              reduce using rule 5
        '+'             reduce using rule 5
        '-'             reduce using rule 5
        '*'             shift, and enter state 11
        '/'             shift, and enter state 12
        ')'             reduce using rule 5
        %eof            reduce using rule 5

State 9

        Factor -> '(' . Exp ')'                             (rule 11)

        let             shift, and enter state 2
        int             shift, and enter state 7
        var             shift, and enter state 8
        '('             shift, and enter state 9

        Exp             goto state 10
        Exp1            goto state 4
        Term            goto state 5
        Factor          goto state 6
```

For each state, the first set of lines describes the *rules* which correspond to this state. A period `.` is inserted in the production to indicate where, if this is indeed the correct production, we would have parsed up to. In state 5, there are multiple rules, so we don't know if we are parsing an `Exp1`, a multiplication or a division (however, we do know there is a `Term` on the parse stack); in state 9, there is only one rule, so we know we are definitely parsing a `Factor`.

The next set of lines specifies the action and state transition that should occur given a token. For example, if in state 5 we process the `'*'` token, this token is shifted onto the parse stack and we transition to the state corresponding to the rule `Term -> Term '*' . Factor` (matching the token disambiguated which state we are in.)

Finally, for states which shift on non-terminals, there will be a last set of lines saying what should be done after the non-terminal has been fully parsed; this information is effectively the stack for the parser. When a reduce occurs, these goto entries are used to determine what the next state should be.

7.2 Interpreting conflicts

When you have a conflict, you will see an entry like this in your info file:

```
State 432

        atype -> SIMPLEQUOTE '[' . comma_types0 ']'         (rule 318)
        sysdcon -> '[' . ']'                                (rule 613)

        '_'             shift, and enter state 60
        'as'            shift, and enter state 16

...

        ']'             shift, and enter state 381
                         (reduce using rule 328)
```

(continues on next page)

(continued from previous page)

```
...
```

On large, complex grammars, determining what the conflict is can be a bit of an art, since the state with the conflict may not have enough information to determine why a conflict is occurring).

In some cases, the rules associated with the state with the conflict will immediately give you enough guidance to determine what the ambiguous syntax is. For example, in the miniature shift/reduce conflict described in *Conflict Tips*, the conflict looks like this:

```
State 13

        exp -> exp . '+' exp0                                   (rule 1)
        exp0 -> if exp then exp else exp .                      (rule 3)

        then            reduce using rule 3
        else            reduce using rule 3
        '+'             shift, and enter state 7
                         (reduce using rule 3)

        %eof            reduce using rule 3
```

Here, rule 3 makes it easy to imagine that we had been parsing a statement like `if 1 then 2 else 3 + 4`; the conflict arises from whether or not we should shift (thus parsing as `if 1 then 2 else (3 + 4)`) or reduce (thus parsing as `(if 1 then 2 else 3) + 4`).

Sometimes, there's not as much helpful context in the error message; take this abridged example from GHC's parser:

```
State 49

        type -> btype .                                         (rule 281)
        type -> btype . '->' ctype                              (rule 284)

        '->'            shift, and enter state 472
                         (reduce using rule 281)
```

A pair of rules like this doesn't always result in a shift/reduce conflict: to reduce with rule 281 implies that, in some context when parsing the non-terminal `type`, it is possible for an `'->'` to occur immediately afterwards (indeed these source rules are factored such that there is no rule of the form `... -> type '->' ...`).

The best way this author knows how to sleuth this out is to look for instances of the token and check if any of the preceding non-terminals could terminate in a type:

```
texp -> exp '->' texp                                   (500)
exp -> infixexp '::' sigtype                            (414)
sigtype -> ctype                                        (260)
ctype -> type                                           (274)
```

As it turns out, this shift/reduce conflict results from ambiguity for *view patterns*, as in the code sample `case v of { x ::  T -> T ... }`.

CHAPTER

EIGHT

TIPS

This section contains a lot of accumulated lore about using Happy.

8.1 Performance Tips

How to make your parser go faster:

- If you are using GHC, generate parsers using the `-a -g -c` options, and compile them using GHC with the `-fglasgow-exts` option. This is worth a *lot*, in terms of compile-time, execution speed and binary size.[4]
- The lexical analyser is usually the most performance critical part of a parser, so it's worth spending some time optimising this. Profiling tools are essential here. In really dire circumstances, resort to some of the hacks that are used in the Glasgow Haskell Compiler's interface-file lexer.
- Simplify the grammar as much as possible, as this reduces the number of states and reduction rules that need to be applied.
- Use left recursion rather than right recursion wherever possible. While not strictly a performance issue, this affects the size of the parser stack, which is kept on the heap and thus needs to be garbage collected.

8.2 Compilation-Time Tips

We have found that compiling parsers generated by Happy can take a large amount of time/memory, so here's some tips on making things more sensible:

- Include as little code as possible in the module trailer. This code is included verbatim in the generated parser, so if any of it can go in a separate module, do so.
- Give type signatures for everything (see *Type Signatures*. This is reported to improve things by about 50%. If there is a type signature for every single non-terminal in the grammar, then Happy automatically generates type signatures for most functions in the parser.
- Simplify the grammar as much as possible (applies to everything, this one).
- Use a recent version of GHC. Versions from 4.04 onwards have lower memory requirements for compiling Happy-generated parsers.
- Using Happy's `-g -a -c` options when generating parsers to be compiled with GHC will help considerably.

[4] omitting the `-a` may generate slightly faster parsers, but they will be much bigger.

8.3 Finding Type Errors

Finding type errors in grammar files is inherently difficult because the code for reductions is moved around before being placed in the parser. We currently have no way of passing the original filename and line numbers to the Haskell compiler, so there is no alternative but to look at the parser and match the code to the grammar file. An info file (generated by the `-i` option) can be helpful here.

Type signature sometimes help by pinning down the particular error to the place where the mistake is made, not half way down the file. For each production in the grammar, there's a bit of code in the generated file that looks like this:

```
HappyAbsSyn<n> ( E )
```

where `E` is the Haskell expression from the grammar file (with `$n` replaced by `happy_var_n`). If there is a type signature for this production, then Happy will have taken it into account when declaring the HappyAbsSyn datatype, and errors in `E` will be caught right here. Of course, the error may be really caused by incorrect use of one of the `happy_var_n` variables.

(this section will contain more info as we gain experience with creating grammar files. Please send us any helpful tips you find.)

8.4 Conflict Tips

Conflicts arise from ambiguities in the grammar. That is, some input sequences may possess more than one parse. Shift/reduce conflicts are benign in the sense that they are easily resolved (Happy automatically selects the shift action, as this is usually the intended one). Reduce/reduce conflicts are more serious. A reduce/reduce conflict implies that a certain sequence of tokens on the input can represent more than one non-terminal, and the parser is uncertain as to which reduction rule to use. It will select the reduction rule uppermost in the grammar file, so if you really must have a reduce/reduce conflict you can select which rule will be used by putting it first in your grammar file.

It is usually possible to remove conflicts from the grammar, but sometimes this is at the expense of clarity and simplicity. Here is a cut-down example from the grammar of Haskell (1.2):

```
exp     : exp op exp0
        | exp0

exp0    : if exp then exp else exp
        ...
        | atom

atom    : var
        | integer
        | '(' exp ')'
        ...
```

This grammar has a shift/reduce conflict, due to the following ambiguity. In an input such as

```
if 1 then 2 else 3 + 4
```

the grammar doesn't specify whether the parse should be

```
if 1 then 2 else (3 + 4)
```

or

```
(if 1 then 2 else 3) + 4
```

and the ambiguity shows up as a shift/reduce conflict on reading the 'op' symbol. In this case, the first parse is the intended one (the 'longest parse' rule), which corresponds to the shift action. Removing this conflict relies on noticing that the expression on the left-hand side of an infix operator can't be an `exp0` (the grammar previously said otherwise, but since the conflict was resolved as shift, this parse was not allowed). We can reformulate the `exp` rule as:

```
exp     : atom op exp
        | exp0
```

and this removes the conflict, but at the expense of some stack space while parsing (we turned a left-recursion into a right-recursion). There are alternatives using left-recursion, but they all involve adding extra states to the parser, so most programmers will prefer to keep the conflict in favour of a clearer and more efficient parser.

8.4.1 LALR(1) parsers

There are three basic ways to build a shift-reduce parser. Full LR(1) (the `L' is the direction in which the input is scanned, the `R' is the way in which the parse is built, and the `1' is the number of tokens of lookahead) generates a parser with many states, and is therefore large and slow. SLR(1) (simple LR(1)) is a cut-down version of LR(1) which generates parsers with roughly one-tenth as many states, but lacks the power to parse many grammars (it finds conflicts in grammars which have none under LR(1)).

LALR(1) (look-ahead LR(1)), the method used by Happy and yacc, is a tradeoff between the two. An LALR(1) parser has the same number of states as an SLR(1) parser, but it uses a more complex method to calculate the lookahead tokens that are valid at each point, and resolves many of the conflicts that SLR(1) finds. However, there may still be conflicts in an LALR(1) parser that wouldn't be there with full LR(1).

8.5 Using Happy with GHCi

GHCi's compilation manager doesn't understand Happy grammars, but with some creative use of macros and makefiles we can give the impression that GHCi is invoking Happy automatically:

- Create a simple makefile, called `Makefile_happysrcs`:

```
HAPPY = happy
HAPPY_OPTS =

all: MyParser.hs

%.hs: %.y
    $(HAPPY) $(HAPPY_OPTS) $< -o $@
```

- Create a macro in GHCi to replace the `:reload` command, like so (type this all on one line):

```
:def myreload (\_ -> System.system "make -f Makefile_happysrcs"
   >>= \rr -> case rr of { System.ExitSuccess -> return ":reload" ;
                          _ -> return "" })
```

- Use `:myreload` (`:my` will do) instead of `:reload` (`:r`).

8.6 Basic monadic Happy use with Alex

Alex lexers are often used by Happy parsers, for example in GHC. While many of these applications are quite sophisticated, it is still quite useful to combine the basic Happy `%monad` directive with the Alex `monad` wrapper. By using monads for both, the resulting parser and lexer can handle errors far more gracefully than by throwing an exception.

The most straightforward way to use a monadic Alex lexer is to simply use the `Alex` monad as the Happy monad:

```
{
module Lexer where
}

%wrapper "monad"

tokens :-
  ...

{
data Token = ... | EOF
  deriving (Eq, Show)

alexEOF = return EOF
}
```

```
{
module Parser where

import Lexer
}

%name pFoo
%tokentype { Token }
%error { parseError }
%monad { Alex } { >>= } { return }
%lexer { lexer } { EOF }

%token
  ...

%%
  ...

parseError :: Token -> Alex a
parseError _ = do
  ((AlexPn _ line column), _, _, _) <- alexGetInput
  alexError ("parse error at line " ++ (show line) ++ ", column " ++ (show column))

lexer :: (Token -> Alex a) -> Alex a
lexer = (alexMonadScan >>=)
}
```

We can then run the finished parser in the `Alex` monad using `runAlex`, which returns an `Either` value rather than throwing an exception in case of a parse or lexical error:

```
import qualified Lexer as Lexer
import qualified Parser as Parser

parseFoo :: String -> Either String Foo
parseFoo s = Lexer.runAlex s Parser.pFoo
```

CHAPTER

NINE

INDICES AND TABLES

- genindex
- search

BIBLIOGRAPHY

[Rekers1992] Parser Generation for Interactive Environments, PhD thesis, University of Amsterdam, 1992.

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

P

R

T

Y