TOM gp

gp Generates Parsers
edition 0, for `gp` version 0.
July 1998

Pieter J. Schoenmakers

Written by Pieter J. Schoenmakers <tiggr@gerbil.org>

1 Overview

GP is a parser generator. The input to `gp` is an annotated LL(1) attribute grammar. From this input, `gp` generates a TOM class that implements a recursive descent parser for that grammar. Each rule of the grammar becomes a method of the parser class.

1.1 Reporting bugs

The latest version of `gp`, this manual, and related goodies is always available from `http://www.gerbil.org/tom/resources/packages.shtml`.

Bugs with `gp` or this manual can be reported to the author, at

`Pieter Schoenmakers <tiggr@gerbil.org>`

or

ir. Pieter J. Schoenmakers
Eindhoven University of Technology
Room EH 9.26
P.O.Box 512
NL-5600 MB Eindhoven
the Netherlands

2 Tutorial

This chapter will introduce the various elements of a `gp` grammar.

2.1 A start

The first example grammar accepts a language of parentheses. The accepted sentences must be well-parenthesized. Examples of valid sentences are '`()`', '`(())(())`', '`(()(()))`', and '' (the empty sentence). Invalid sentences include '`)`', '`(`', and '`((()())`'.

This is the grammar for the language of parentheses:

```
%class ex1

language: expr*;
expr: '(' expr* ')';
```

The first line defines the class name of the parser that `gp` will create. Incidentally, that name is also used as the name of the directory in the distribution of this manual that contains this example. To see this example in action, just *cd* to the directory '`ex1`' and type *make*, followed by *./ex1*.

The next line defines a rule of the grammar. Since it is the first rule in the grammar, it is the top rule and it defines the language that will be accepted by the grammar.

```
language: expr*;
```

This rule defines the expansion of the non-terminal `language` to be `expr*` . The rule is terminated by a semicolon (`;`). The '`*`' character is one of several postfix operators that modify how the preceding entity is to be matched. Without any operators, a terminal or non-terminal must occur in the input exactly once. The operators modify this requirement:

`?` The entity is optional (<= 1).

`*` The entity may occur any number of times (>= 0).

`+` Like `*`, but at least once (>= 1).

It is clear that '`a*`' is identical to '`a+?`', i.e., 'zero or more times '`a`'' is equal to 'optionally once or more often '`a`''.

The next line in the `ex1` example defines the rule for `expr`:

```
expr: '(' expr* ')';
```

Thus, an `expr` is an opening parenthesis (`(`), followed by zero or more `exprs`, followed by the closing parenthesis (`)`). The juxtaposition of the elements signify the implicit concatenation operator.

Now all non-terminals have been defined, our grammar is complete. An example of using this parser by running '`ex1`' (each input line is followed by a *^D* to indicate end of file):

```
$ ./ex1
()
$ ./ex1
)
stdin:1: error at ')'
```

```
$ ./ex1
(
stdin:1: error at EOF, expected token ')'
$
```

2.2 Semantic actions

Parsers are never created just to accept sentences from some language; they will always *do something* in response. In a `gp` grammar, such actions can be added using '`{...}`'. For example, to let the parser from the previous example (`ex1`) note on its output each time it has just read an opening parenthesis ('`(`'), we could modify the `expr` rule like this:

```
expr: '(' { [[[stdio out] print "read another `('"] nl]; } expr* ')';
```

and end up with something quite unreadable. To increase readability, especially when we need to add a lot of semantic actions, we adopt the coding style of putting the actions clearly indented and on a separate line:

```
expr:
          '('
                {
                  [[[stdio out] print "read another `('"] nl];
                }
          expr* ')'
        ;
```

2.3 Tokens

In the first example, only plain characters were used as tokens. Normally, however, tokens will consist of more than a single character. For example, numbers larger than 9 will take at least two characters. To tell the parser about such tokens, the '`%token`' directive is used to declare the names of the tokens. By convention, these names are in upper-case, whereas the non-terminal names are lower-case.

In the following example, the grammar accepts sentences with any number of '`NUMBER`' tokens. Note that it is up to the lexer to recognize such tokens in the input and return the proper token value to the parser.

```
%class ex2

%token NUMBER

language: NUMBER*;
```

The lexer for this parser must know about the `NUMBER` token as well. For our parser class '`ex2`', `gp` also generates a class '`ex2Defines`' that contains the definitions of the token values. Our lexer simply inherits from that class to access these values (see '`ex2/Lexer.t`').

2.4 Token semantics

We now look at the semantics of tokens. The semantic value of a '`(`' is void: the fact that it is an open parenthesis is all the information it contains. However, when reading a number, the value of the number is important. To illustrate this, the next example is the language of numbers; the parser will add the numbers together and display the result at the end of a sentence.

A first go at the parser looks like this:

```
%class ex2

%token NUMBER

language
                {
                  int value;
                }
        :
          [NUMBER
                {
                  value += ...;
                }
          ]*
                {
                  [[[stdio out] print ("result: ", value)] nl];
                }
        ;
```

The semantic block '{ int value; }' between the name of the non-terminal ('language') and the colon at the start of the rule, is visible in the whole rule, and therefore the designated place to declare local variables.

The '*'-operator is preceded by two entities between '[' and ']'. These brackets group the entities in between them and make the operator apply to the whole group. Every time a NUMBER is matched, the action will also be executed.

One thing remains to be defined: how is the value of the NUMBER token retrieved from the lexer, i.e., what must be filled in at the place of the '...'? We can make our lexer provide a method

```
int
  number_value;
```

which we can invoke to retrieve the value of the most recently read NUMBER token. In this case, the action becomes

```
value += [lex number_value];
```

'lex' is an instance variable of the parser, of the type gps.Lexer. Obviously, this class does not implement the 'number_value' method. We could add it as an extension, but prefer to employ a subclass instead. To indicate this class to the TOM compiler, the lex instance variable must be redeclared. For the purpose of (re-)declaring instance variables, gp provides the '%instance' directive, which must be placed following the '%class' directive.

```
%instance
{
  <doc> Our lexer is an {ex2.Lexer}.  </doc>
  redeclare ex2.Lexer lex;
```

```
}
```

This completes our number-adding parser. The directory 'ex2' contains the full source, including a lexer that can recognize integer numbers, which it returns as NUMBER tokens. Some sample runs:

```
$ echo | ./ex2
result: 0
$ echo 1 2 3 4 | ./ex2
result: 10
$ echo -1 | ./ex2
result: 0
stdin:1: error at '-'
$ ./ex2
40
2
result: 42
```

2.5 Alternatives

No choice is not good. It is very common for a non-terminal to be expandable in several directions. In the third example, we expand the language from the first example to the language of parentheses, square brackets, and braces. The new grammar becomes:

```
%class ex3

language: expr*;
expr: '(' expr* ')'
    | '[' expr* ']'
    | '{' expr* '}'
    ;
```

The vertical bar ('|') divides the various alternatives for the expansion of an `expr`. Now '()', '[]', and '{}' are all valid expansions of `expr`. Note that the (implicit) concatenation of entities has a higher priority than the alternatives: it is not necessary to group each alternative, like in `'['(' expr* ')']'`.

2.6 Synthesized attributes

So far, rules were like procedures: you invoke them and after some time they return. However, GP employs an attribute grammar, meaning that rules can be passed attributes and also return them. In this section, we'll look at an infix-expression parser, in which each rule that is responsible for parsing a subexpression will return the result of evaluating that part of the expression.

The calculator repeatedly reads an expression and prints the result of evaluating the expression. The top rule of the grammar will look like this:

```
%class ex5

%token NUMBER
```

```
grammar
                {
                  int result;
                }
          :
            [expr (result) ';'
                {
                  [[[stdio out] print result] nl];
                }
            ]*
 ;
```

The non-terminal `expr`, which has not been defined yet, is used as in

```
expr (result)
```

indicating that the `expr` rule employs one attribute. We need to look at the definition of `expr`, to see what kind of attribute this actually is. The `expr` rule starts like this:

```
expr (<int v)
  ...
```

This indicates that `expr` has one attribute, '`v`', of type `int`. The prefix '`<`' means that it is a synthesized attribute, i.e., a value for the attribute is returned to the invoking rule.

In the output of `gp`, the `expr` rule will be implemented by the following method:

```
int (v)
  expr;
```

Thus, the `expr` method returns an `int` corresponding to the local variable '`v`'. Setting the variable `v` within the rule will affect the value that is returned. Similarly, a rule with multiple synthesized attributes is implemented with multiple return values.

Also in the code generated by `gp`, the use of the `expr` rule becomes:

```
result = [self expr];
```

The full `expr` rule looks like this:

```
expr (<int v)
                {
                  int right;
                }
          :
            term (v)
            ['+' term (right)
                {
                  v += right;
                }
            |'-' term (right)
                {
                  v -= right;
                }
            ]*
          ;
```

In words: an `expr` starts with a `term`, followed by the addition or subtraction of zero or more `term`s. Similar to an `expr`, a `term` has one synthesized attribute. The value it returns is stored in the local variable `v`, which happens to be the name of the value that the `expr` rule returns: if the `term` is not followed by an addition or subtraction, the value returned by the `term` will be returned by the `expr`.

When `expr` is to parse the expression '`1 + 2`', the invocation of `term` will digest the '`1`' and return 1. This 1 will be stored in `v`. Since it is followed by a '`+`', the first branch of the alternative that follows will be taken. The `term` will be invoked again to parse what remains, i.e. '2'. The value that it returns is stored in the variable `right`, and the action that follows will be executed, which will add `right` to `v`, making `v` equal to 3. The end of the input has been reached and the `expr` rule will return the value of `v`: it has successfully parsed the input and correctly returned the result: 1 + 2 is 3.

The parser is still not complete, with the non-terminal `term` not yet defined. What an `expr` does with addition and subtraction, a `term` does with multiplication and division. Because the operators are handled by two rules and because the `expr` depends on `term`, the operator priorities are handled automatically and '`1 + 2 * 3`' will produce 7 and not 9.

```
term (< int value)
                {
                  int left;
                }
        : atom (left)
          ['*' atom (value)
                {
                  left = left * value;
                }
          |'/' atom (value)
                {
                  left = left / value;
                }
          ]*
                {
                  value = left;
                }
        ;
```

Note how `term` uses a different approach than `expr` to accumulate the result and hence must set the correct return value after it has finished matching the input.

The definition of the `atom` completes the parser. An `atom` is a number, a parenthesized expression, or a negated `atom`.

```
atom (< int value)
        :
          NUMBER
                {
                  value = [lex number_value];
                }
        | '-' atom (value)
                {
```

```
                value = -value;
              }
        | '(' expr (value) ')'
        ;
```

2.7 Inherited attributes

Inherited attributes are attributes that are *passed to* a rule, as opposed to synthesized attributes, which are *returned from* a rule.

As an example, we rewrite example 5 to handle the operator priorities within a single rule: the priority of the expression surrounding the subexpression being handled is passed as an attribute. For this purpose, we define some constants to give names to the various priorities.

```
%class ex6

%instance
{
  redeclare Lexer lex;

  const PRIO_EXPR = 0;
  const PRIO_ADDSUB = 1;
  const PRIO_MULDIV = 2;
  const PRIO_UNARY = 3;
}

%token NUMBER
```

The `expr` rule is modified to accept a the priority of the expression that it is to parse. When, for example, an subexpression must be parsed at priority `PRIO_ADDSUB`, a multiplication must be parsed before the subexpression is completed. If, however, another addition is encountered, the subexpression is finished (since parsing the addition before returning would imply right-associativity, which is not the desired semantics).

The `expr` rule becomes:

```
expr (> int priority, < int value)
                {
                  int right;
                }
        :
          atom (value)
          [
                        { if (priority >= PRIO_ADDSUB) return; }
            '+' expr (PRIO_ADDSUB, right)
                        { value += right; }
          |
                        { if (priority >= PRIO_ADDSUB) return; }
            '-' expr (PRIO_ADDSUB, right)
                        { value -= right; }
          |
```

```
                        { if (priority >= PRIO_MULDIV) return; }
                '*' expr (PRIO_MULDIV, right)
                        { value *= right; }
              |
                        { if (priority >= PRIO_MULDIV) return; }
                '/' expr (PRIO_MULDIV, right)
                        { value /= right; }
              ]*
            ;
```

A top-level invocation of the `expr` rule becomes

```
expr (PRIO_EXPR, result)
```

A closer look at the branches in the `expr` reveals a very common structure:

```
            { if (priority >= PRIO_ADDSUB) return; }
'+' expr (PRIO_ADDSUB, right)
```

A particular branch will be selected when the current token matches (in this case, it would be a '`+`'). However, when the priority of the next operator is not higher than that of the expression already being parsed (passed as an attribute in `priority`), the `return` is executed and the operator token still remains to be taken from the input, for the invoking rule to digest it and correctly accept an expression.

2.8 Methods

We already know how to use `{}` to include literal TOM code in a rule. When you want to add code to the parser class outside of a rule, things are different. Since such code can only be a method, such methods can be added as an extension to the parser class, using another source file. However, this is not all that convenient, and extending a class within the same unit is not considered to be elegant.

`gp` provides the compound `%%...%%` construct to define methods. At any top-level location in the grammar, such a compound may be used to declare or define methods. Before the `%instance` has been seen, any method defined is a class method. After the `%instance`, the method is an instance method.

Example '`ex7`' starts with a class method `void note int`.

```
%class ex7

%%
<doc> Print the {tok}.  This is a class method.  </doc>
void
  note int tok
{
  [[[stdio out] print ("next token: ", byte (tok))] nl];
}
%%
```

Although the instance definition can remain empty, a `%instance` is needed to indicate that the next method definition between `%%...%%` is an instance method.

```
%instance

%%
<doc> Pass to the same method of our class.  </doc>
void
  note int tok
{
  [isa note tok];
}
%%
```

The grammar itself is rather silly, but demonstrates that the method that we defined can be invoked. The `token` argument is an instance variable of the `gps.Parser` class; it holds the current token.

```
language: expr*;
expr:
                {
                  [self note token];
                }
        [ '(' expr* ')'
        | '[' expr* ']'
        | '{' expr* '}'
        ]
        ;
```

3 Grammar

This chapter defines the input to gp. The grammar described by it must be a LL(1) grammar; gp will report an error if it is not; it will not attempt to rewrite the grammar into an equivalent LL(1) grammar.

3.1 Comments

gp accepts the same kinds of comments as TOM.

```
// This line is a C++-style comment.
// Likewise.  It will trigger a warning.
/* This is multi-line
   C-style comment.  */
<doc> This is TOM-style HTML-like comment.  </doc>
```

3.2 %class

3.3 %instance

3.4 %token

3.5 %predef

3.6 %%

3.7 Rules

4 Unit gps

The classes generated by `gp` depend on the TOM unit `gps`. This name is short for '`gp` skeleton'. The `gps` unit contains the following classes:

`Parser`
: This is the superclass of every `gp`-generated parser. It contains the machinery for reporting errors and warnings, interacting with the lexer, tracing a parser's operation, etc.

`Lexer`
: A fully deferred, stateless class (i.e., *interface* or *behavioural class*) that defines the interface to a lexer, as employed by a `Parser`.

`SkeletonLexer`
: A subclass of `Lexer`, this class provides a skeleton for real lexers, including important instance variables and part of the interface defined by `Lexer`.

`StreamLexer`
: Subclass of `SkeletonLexer`, the `StreamLexer` is provided as a handy superclass for lexers that read from an `InputStream`, as will almost all lexers do.

`FileElement`
: A bare class that knows how to report its location (line number). It is intended to be used as a superclass of semantic entities that are built from reading the input, so that reporting errors for them through the parser is easy (since the parser knows how to report an error location for `FileElement` objects).

Table of Contents

1 Overview **1**

1.1 Reporting bugs 1

2 Tutorial **3**

2.1 A start 3
2.2 Semantic actions 4
2.3 Tokens 4
2.4 Token semantics 4
2.5 Alternatives 6
2.6 Synthesized attributes 6
2.7 Inherited attributes 9
2.8 Methods 10

3 Grammar **13**

3.1 Comments 13
3.2 %class 13
3.3 %instance 13
3.4 %token 13
3.5 %predef 13
3.6 %% 13
3.7 Rules 13

4 Unit gps **15**