pacc - a compiler-compiler

for version v0.3, 30 July 2016

Tobold J. Goodwin (toby@paccrat.org)

This manual is for GNU pacc (version v0.3, 30 July 2016), which is a parser generator.

Table of Contents

1 Parsing basics

A *parser* is a computer program that interprets the utterances of a *computer language*. A computer language is more properly known as a *formal language*. (Formal languages are almost entirely *un*like natural languages—*English*, *Pŭtōnghuà*, *Castellano*, etc. It is usually a mistake to make analogies or comparisons between human languages and computer languages.)

A computer language is a (usually) infinite set of *utterances* or *sentences*. Each utterance is itself a finite string of symbols drawn from a finite *alphabet*. Although the set of all possible utterances is infinite, a useful computer language can be described by a finite (and usually quite short) *grammar*. For example, consider the computer language which consists of all the positive even numbers, written as decimals. This makes up an infinite set of utterances, written with an alphabet of ten symbols (‘`0`’, ‘`1`’, . . . , ‘`9`’). Not every combination of those symbols is a valid utterance of this language, but we don't have to list all the ones that are. The language can be described very briefly: any combination of those symbols, provided the last one is one of (‘`0`’, ‘`2`’, ‘`4`’, ‘`6`’, ‘`8`’).

Computer languages vary enormously in complexity: the most elaborate are high-level programming languages. At the bottom end, the address box in a web browser understands an extremely simple computer language, the majority of whose utterances begin ‘`http://`’! Somewhere in between is the language of cells in a spreadsheet, in which one can say things like ‘`=A2+3*sum(C7:C9)`’.

Pacc is a *parser generator*. That means that we write a compact description of a computer language, invoke `pacc` with that description as its input, and pacc writes for us a parser for that language. The output of pacc is a C function, which we can use to build a complete program.

When we run a program that includes a pacc-built parser, the program will procure some input from somewhere—in most cases it will be a file, but it could be the command line, or a GUI input—and hand that input to the parser. The parser now has a couple of jobs to do.

First, it needs to decide if the input is in fact a valid utterance of the language described by the grammar. For example, if the language is simple arithmetic expressions, then ‘`2 * (3 + 4)`’ presumably *is* a valid utterance, whereas ‘`2 + * )( 3 4`’ is *not*. If the input is erroneous (not a valid utterance of the described language), then the parser needs to indicate how much it could parse before it encountered an error. If we are parsing a programming language, our utterances may be thousands of lines long, so good error-handling is vital.

Secondly, if the input was valid, the parser needs to discern its meaning. In most cases, this will involve building an *Abstract Syntax Tree* AST that captures the meaning of the input in a way that can easily be manipulated by the rest of the program. (In our first examples, though, we won't be building trees.)

2 Getting started

Let's actually make a pacc parser. We'll start with about the simplest possible language I can conceive of. In this language, there are just two valid utterances: **yes** and **no**. The "meaning" of **yes** will be **1**; **no** will mean **0**. This can be expressed in a pacc grammar like this.

```
Start <- "yes" {1} / "no" {0}
```

We'll worry about the details of exactly how that expresses the language we described later on (although you can probably get the general idea). For now, we want to turn this into an actual program, so the first thing to do is to put that grammar into a file named `bool.pacc`.

Next, we'll need to supply the rest of the program. We'll keep it very simple. Here's `main.c`.

```
#include <stdio.h>
#include <string.h>

#include "bool.h"

int main(int argc, char **argv) {
    int result;

    if (argc != 2) {
        fprintf(stderr, "one argument please\n");
        return 1;
    }

    if (pacc_wrap("arg", argv[1], strlen(argv[1]), &result))
        printf("parsed with value %d\n", result);

    else
        return 1;

    return 0;
}
```

To build the program, we first need to invoke `pacc` on the grammar definition. This will output `bool.c`. We specify the `-d` switch to `pacc` so that it will also output `bool.h` which we included in our program, and supplies the declaration of `pacc_wrap()`:

```
pacc -d bool.pacc
```

Then simply compile as normal. The output of pacc is valid C99, but not C89, so you will probably need to explicitly invoke a C99 compiler. (This is definitely an area that needs further study. While I'm not greatly swayed by the argument that pacc should be compatible with a 25 year old standard, it is unfortunate that currently if you hand its output to gcc you get an obscure error message.)

```
c99 -o bool main.c bool.c
```

And here's what the program looks like in action:

```
$ bool yes
parsed with value 1
$ bool no
parsed with value 0
$ bool foo
arg:1:1: expected Start
```

Which is what we wanted!

3 Pacc basics

A pacc grammar consists of one or more *rules*. Let's look at our first example again.

```
Start <- "yes" {1} / "no" {0}
```

There's just one rule in this tiny grammar. Every rule starts with a name, and the name of this rule is '`Start`'.

After the name, a left arrow introduces the *definition* of the rule. In this case, we've made a left arrow out of two old-fashioned ASCII symbols, '`<`' and '`-`', but it's also permitted to use the Unicode character '←' (U+2190 LEFTWARDS ARROW). You can also use a plain old equals sign '`=`' if you prefer.

A pacc parser works by *matching* the input to the rules in the grammar. The process starts with the first rule (in this case it's the only rule), and the first thing in the definition of that first rule. In the case of '`Start`', the first thing in the definition is '`"yes"`'. The quote marks mean that this is a *literal string*. For a successful match, the input must be exactly the same characters as are in the literal.

Let's suppose that on this occasion the input is '`no`'. So the parser can't match the input against '`"yes"`' in the rule. But in this rule there is an *alternative*, and the parser will try to match that next. The '`/`' character separates alternatives, so the next alternative starts with another literal string: '`"no"`'. This does match our example input.

So having matched '`"no"`' the parser then finds '`{0}`'. The braces indicate that this is a *semantic expression*: a hunk of C code (albeit a very tiny one in this case!). Because it is in the alternative that matched, it will be evaluated to give a value to the rule. And the value of the start rule is the overall value of the parse.

4 Alternatives

> An absolutely crucial point to understand about pacc grammars is that alternatives are tried **in order**. The first alternative which matches successfully is the one that "wins", and later alternatives aren't considered at all.

Let's go multi-lingual. (There are more elegant ways to write this, which we'll come to soon.)

```
Start <- "yes" {1} / "oui" {1} / "no" {0} / "non" {0} # wrong!
```

That may look like it should work, but this is what happens when we try it:

```
$ french0 oui
parsed with value 1
$ french0 no
parsed with value 0
$ french0 non
arg:1:3: expected end-of-input
```

What went wrong there? The problem is that the third alternative '`"no"`' matches the first two characters of the input '`non`', and so that becomes the winning alternative, even though that leads to an overall failure. Pacc parsers never "back up" once an alternative has completely matched.

For this simple grammar, the obvious fix is simply to swap the order of these two alternatives. We'll see similar problems, and other fixes, later on.

```
Start <- "yes" {1} / "oui" {1} / "non" {0} / "no" {0}
```

This grammar works as intended:

```
$ french1 no
parsed with value 0
$ french1 non
parsed with value 0
```

This behaviour is the fundamental difference between PEGs (the style of grammars that pacc uses) and CFGs (the more traditional alternative), so it may take you a while to get used to it. (This paragraph needs rewriting and expanding.)

5 Rules

Here's where we've got to:

```
Start <- "yes" {1} / "oui" {1} / "non" {0} / "no" {0}
```

We can improve this by splitting things up and removing the repetition.

```
Start <- Yes ↦ 1 / No ↦ 0
Yes :: void <- "yes" / "oui"
No <- "non" / "no"
```

Don't, at the moment, worry about '`:: void`' on the second line—we'll explain what this means shortly. The important point to notice is that a grammar can have more than one rule. The first rule is always the start rule, where parsing begins.

Each rule has a name, in this grammar there are 3 rules named '`Start`', '`Yes`', and '`No`'. Rule names are formed in the same way as C identifiers: they must start with a letter or underscore, and continue with letters, underscores, or digits. I often follow the convention of starting rule names with capital letters as it helps them to stand out, but you don't have to.

The fundamental building blocks of pacc are called *matchers*. We've already met one kind of matcher: a literal string like '`"yes"`'. A matcher occurs in the definition of a rule, and it matches something in the input. In the case of a literal string, it matches just that one string.

Another kind of matcher is a *call matcher*. One rule can call another rule, simply by naming the called rule in its definition. As you might expect, when you call a rule by name, that matches whatever the called rule matches. The definition of the '`Start`' rule contains two call matchers: the first calls the rule '`Yes`' and the second calls the rule '`No`'. You can call a rule before it has been defined, so long as the definition occurs in the same pacc grammar.

There are only two other kinds of matcher, which we'll get to soon.

6 Types

Every pacc rule has a *type*, which can be almost any C type. The type of a rule appears in its definition, directly after the name introduced by '`::`'. (In many ways, I would have preferred to prefix types to names like C itself, but then we'd have to terminate rules, say with semicolons, and that appealed even less.)

The rules for types are as follows.

1. If a rule has an explicit type, that is its type; otherwise
2. the rule has the same type as the previous rule; unless
3. it is the first rule, in which case the default type is '`int`'.
4. Every path through a rule must include exactly one expression with the type of the rule; unless
5. the type of the rule is '`void`', in which case no expressions are allowed.

Rule 3 explains how we have got away so far without explicit type declarations—in their absence, everything defaults to '`int`'. Let's revisit our example yet again.

```
Start :: int <- Yes ↦ 1 / No ↦ 0
Yes :: void <- "yes" / "oui"
No <- "non" / "no"
```

The first rule '`Start`' has type '`int`', this time we've made that explicit. The second rule '`Yes`' has type '`void`', again by explicit declaration. The third rule '`No`' has no explicit type, so by rule 2 its type is '`void`'.

> **Warning:** The type of an implicitly-typed rule depends on where it is in the grammar. Rearranging rules can therefore change the meaning of a grammar!

Arguably this is unfortunate (perhaps there should be a '`strict`' mode where every rule must have an explicit type?) but in practice most pacc grammars have only a few different types, so implicit typing saves a lot of, errm, typing.

Any C type that can be constructed with words and the '`*`' character can be used as a pacc type. This includes pointers, structures, and unions, but not vectors (arrays) or function types. You could use a '`typedef`' for those if you really wanted to, and include it (directly or more likely by '`#include`') in the pacc preamble.

7 Character classes

For our next example, let's consider how we could parse a decimal number. We'll start by parsing a single digit, and we already know one way we could do that:

```
Digit <- "0" ↦ 0 / "1" ↦ 1 / "2" ↦ 2 / "3" ↦ 3 / "4" ↦ 4 /
  "5" ↦ 5 / "6" ↦ 6 / "7" ↦ 7 / "8" ↦ 8 / "9" ↦ 9
```

This is tedious and error-prone, but it's the best we can do with the matchers we've seen so far. Another matcher available in pacc is the *character class*: square brackets enclose a set of characters to be matched, so the character class '`[0123456789]`' matches a single decimal digit. To compress it even further, we can write '`[0-9]`'. The hyphen stands for all the characters between the two characters it separates.

But while solving one problem—concisely matching any of a set of characters—we have introduced another: how do we write a useful semantic expression to go with a character class matcher?

```
Digit <- [0-9] -> { /* what goes here? */ }
```

Now that we have just one alternative that will match any digit, what expression can we use that will give us the right value? We need some way to refer back to input that we are matching. We'll see how we do that in the next section.

As well as normal character classes, pacc supports *negated character classes*. These are written with a caret '`^`' as the first character. For example, '`[^)]`' matches any single character *except* a closing parenthesis.

Should you need to write a character class that includes the literal character '`^`', simply ensure that it is not the first character in the class. To match a literal hyphen, write it as the first or last character in the class. To match a literal '`]`', write it as the first character. For example: the character class '`[~^]`' matches either a tilde or a caret; '`[-_]`' matches a hyphen or an underscore; and '`[][]`' matches either an opening or a closing bracket.

If you are familiar with character classes in regular expressions, pacc's character classes are broadly similar. Note, however, that pacc is not aware of locales. In pacc, a hyphenated range only ever stands for all the characters with Unicode code points in the (inclusive) range between those of the two named characters. (Believe it or not, in at least some versions of GNU `grep`, in at least some locales, the character class '`[a-z]`' matches all lower case letters, and also all upper case letters... except '`Z`'.) Nor does pacc support named classes (such as '`[:alpha:]`'). See Chapter 17 [Semantic guards], page 20, if you need to do something like this.

8 References

In any semantic expression, you can write '`ref()`' which is a *reference* to whichever part of the input is matched by the current rule. The type of '`ref()`' is '`ref_t`', an opaque type. A number of helper functions exist to do useful things with '`ref_t`' values. In this case, we can utilize the helper function '`ref_0()`' which returns the first byte of a '`ref_t`' value.

```
Digit <- [0-9] -> { ref_0(ref()) - '0' }
```

9 Operators

So now we can concisely parse a single decimal digit: we match the digit with a character class, and use a reference to the input to extract its value. To extend this to decimal integers, we can use a *repetition operator*. Pacc supports three repetition operators: '`?`', '`*`', and '`+`'. They are postfix operators, that is to say the operator follows its operand. The operand can be a matcher, or a more complicated expression in brackets.

The repetition operators in pacc have the same meanings as they do in regular expressions.

'`?`' Matches *zero or one* occurrences of its operand, or to put it another way, it makes its operand optional.

'`*`' Matches *zero or more* occurrences of its operand. Matches as many times as possible.

'`+`' Matches *one or more* occurrences of its operand. Matches as many times as possible.

A decimal number consists of one or more digits, so the correct choice of repetiton operator for this case is '`+`'.

```
Decimal <- [0-9]+ -> { atoi(ref_str()) }
```

We are using the '`ref_str()`' function which returns a newly allocated, NUL terminated string containing a copy of everything that the current rule matches. And '`atoi()`' is from '`<stdlib.h>`' of course. (One snag with this example is that pacc will happily match an arbitrarily long integer, whereas '`atoi()`' is limited to integers that will fit in an '`int`', but solving that issue is outside the scope of this tutorial. Possible options might include the error-checking available in '`strtol()`', or to use an arbitrary-precision integer package which provides its own conversion function, for example '`gmp_sscanf`'.)

10 Binding values to names

We have seen how we can extract an integer from pacc's input into a C '`int`' value. To do much more, we need to be able to bind a name to that value so that we can refer to it in expressions. In pacc, we use the '`:`' operator, and write '`name:something`' to bind '`name`' to the value of '`something`'.

So what are the type and value assigned to '`name`'? There are two separate cases. If '`something`' is the name of a rule, then '`name`' has the same type as that rule, and the value of evaluating that rule. In all other cases, '`name`' has the type '`ref_t`' and its value is everything matched by '`something`'. For example, if there is a rule '`Number :: int`', then after '`n:Number`', the type of '`n`' is '`int`', but after '`n:Number*`' or '`n:(Number)`' the type of '`n`' is '`ref_t`'.

Let's expand our example to do simple additions.

```
Sum <- d:Decimal "+" s:Sum -> { d + s }
     / d:Decimal -> d
Decimal <- [0-9]+ -> { atoi(ref_str()) }
```

We are using recursion. The base case is the second alternative: a '`Sum`' can be simply a '`Decimal`'. In this case, we bind '`d`' to the value returned by the '`Decimal`' rule, and that value is then returned as the value of the '`Sum`' rule. For the recursive case, we have a '`Decimal`', followed by a literal plus sign, followed by the recursive call to '`Sum`'. We bind '`d`' to the '`Decimal`', and '`s`' to the value returned by the recursive call. The overall value returned in this case is the sum of '`d`' and '`s`'.

A name bound by the '`:`' operator is in scope from its binding to the end of the alternative in which it occurs. Thus, the two occurences of '`d`' in the '`Sum`' rule are in different scopes.

Note that the second alternative in '`Sum`' is a prefix of the first alternative. This means that it is imperative to write the longer expression first, just like in the '`"non" / "no"`' example (see Chapter 4 [Alternatives], page 4).

It is fairly common for one or more alternatives in a rule simply to call another rule and return its result. As a shorthand, pacc allows you to omit the binding and expression for this case. Thus the '`Sum`' rule can be written like this.

```
Sum <- d:Decimal "+" s:Sum -> { d + s }
     / Decimal
```

11 Expressing precedence

Now let's expand our calculator example to include multiplication. This introduces the notion of *precedence*. Multiplication is considered to have higher precedence than addition. So in an expression such as '`2+3*4`', the multiplication is performed first, and the answer is '`14`' (not '`20`').

Some parser generators have explicit notation to express precedence, but in pacc we do it simply by using rules. Here's how we add multiplication to the calculator.

```
Sum <- p:Product "+" s:Sum -> { p + s }
    / Product
Product <- d:Decimal "*" p:Product -> { d * p }
    / Decimal
Decimal <- [0-9]+ -> { atoi(ref_str()) }
```

Simply because of the way the rules are set out, pacc is obliged to perform the higher precedence '`*`' operator before the lower precedence '`+`' operator: it needs a '`Product`' before it can begin evaluating a '`Sum`'.

Of course, as soon as we have precedence, we want to offer the option to override it with brackets. That's easy to implement too.

```
Sum <- p:Product "+" s:Sum -> { p + s }
    / Product
Product <- t:Term "*" p:Product -> { t * p }
    / Term
Term <- Decimal / "(" s:Sum ")" -> s
Decimal <- [0-9]+ -> { atoi(ref_str()) }
```

Again, the higher precedence operation (in this case bracketing) appears lower down the grammar, and everything works out. This grammar can parse '`2+3*4`' with a value of 14, and '`(2+3)*4`' with a value of 20.

12 Handling whitespace

So far, our example grammar insists that everything is squashed together. Normally, we want to allow arbitrary whitespace to appear wherever it makes sense. That usually means between the lexical elements that make up the language. For example, we would like to be able to write '`12 + 34`' or '` 12 +34`' as well as '`12+34`'. Generally, we don't allow whitespace to appear *inside* those lexical elements, so we don't want to accept '`1 2 + 3 4`'.

There are two steps. The first is to define a rule that matches whitespace. Normally it will match at least '`[ \t\n]*`', that is, zero or more occurences of space, tab, and newline. If the language being defined includes comments, they will usually be matched by this rule too (reflecting the fact that comments are considered equivalent to whitespace by most languages: they separate lexical elements). Conventionally the whitespace-matching rule is given the unobtrusive name '`_`' (a single underscore), although you can of course choose any name you like.

The second step is to modify the grammar to use the whitespace rule. We will need to define a separate rule for each lexical element in our grammar. Each such rule will match the element in question, followed by our whitespace rule. For example, '`+`' is a lexical element in our calculator's grammar, so we will define the rule '`Plus <- "+" _`' to match it. Also, the start rule will need to permit whitespace at the beginning of an input. Here's what it looks like.

```
Expression <- _ s:Sum -> s
Sum <- p:Product Plus s:Sum -> { p + s }
       / Product
Product <- t:Term Star p:Product -> { t * p }
         / Term
Term <- Decimal / lBrace s:Sum rBrace -> s
# lexical elements
Decimal <- [0-9]+ _ -> { atoi(ref_str()) }
Plus :: void <- "+" _
Star <- "*" _
lBrace <- "(" _
rBrace <- ")" _
_ <- [ \t\n]*
```

This simple example demonstrates another important and powerful feature of PEG parsing: there is no need for a separate "lexer" to assemble characters into lexical tokens. Pacc can do both jobs, and a single grammar specification describes the entire language, from input characters, through lexical elements, all the way up to complete programs.

13 Left recursion

An obvious next step for our calculator would be to add subtraction. Unfortunately, this is not as simple as you'd hope. Let's rewind to a simpler grammar, our first adder, which looks like this:

```
Sum <- d:Decimal "+" s:Sum -> { d + s }
    / Decimal
Decimal <- [0-9]+ -> { atoi(ref_str()) }
```

Implementing subtraction looks simple; add another alternative to the '`Sum`' rule, like this.

```
Sum <- d:Decimal "+" s:Sum -> { d + s }
     / d:Decimal "-" s:Sum -> { d - s }
     / Decimal
```

This correctly evaluates '`3-2`' giving the answer 1. Unfortunately, if we ask it to evaluate '`3-2-1`', it gives the answer 2, where we were expecting 0. It has made subtraction right associative, so '`3-2-1`' is evaluated as '`3-(2-1)`'. In everyday life, subtraction is considered left associative, and '`3-2-1`' should mean the same as '`(3-2)-1`'.

In a CFG, we could solve the problem by rearranging the rule to look for a Sum first, something like this. (Since addition is commutative, it doesn't matter whether or not we rearrange the first branch of the rule.)

```
Sum <- d:Decimal "+" s:Sum -> { d + s }
     / s:Sum "-" d:Decimal -> { s - d }
     / Decimal
```

But this won't work in a PEG. When we try to compile it, pacc tells us:

```
pacc: fatal: left recursion in rule 'Sum'
```

The second alternative in this rule effectively says: "To match a Sum, first try to match a Sum...". That's an infinite loop, and pacc won't accept it. The exact rule is that each alternative of each rule must make some progress—consume at least one character from the input—before calling itself, whether that call is made directly, or through another rule.

Getting back to subtraction, I'm afraid the sad news is that implementing a left associative operator in the current version of pacc is very difficult. Left recursion is, without doubt, the Achilles' heel of PEG parsing, and there are many ideas about how to handle it better. I have my own thoughts, which may see the light of day in a future release.

One option that works with current versions of pacc is to build a list, then reverse it. Here's a self-contained example that does just that, and can correctly subtract three integers. Whether this could sanely be extended to a more complete expression parser, such as those found in typical programming languages, I do not know.

```
{
  #include <stdio.h>

  /* A future version will avoid needing this predeclaration. */
  int pacc_wrap(const char *, char *, size_t, int *);

  struct action { char op; int val; };
  struct action *stack = 0, *sp = 0, *alloc = 0;

  void push(char op, int x) {
    if (sp == alloc) {
      int s = alloc - stack;
      int n = 2 * s + 1;
```

```
      stack = realloc(stack, n * sizeof *stack);
      if (!stack) pacc_nomem();
      alloc = stack + n;
      sp = stack + s;
    }
    sp->op = op; sp->val = x;
    ++sp;
  }

  int eval(void) {
    int a;
    while (--sp >= stack) {
      switch (sp->op) {
      case ' ': a = sp->val; break;
      case '+': a += sp->val; break;
      case '-': a -= sp->val; break;
      }
    }
    return a;
  }

  int main(int argc, char **argv) {
    int result;
    if (argc != 2) {
      fprintf(stderr, "one argument please\n");
      return 1;
    }
    if (pacc_wrap("arg", argv[1], strlen(argv[1]), &result))
      printf("parsed with value %d\n", result);
    else
      return 1;
    return 0;
  }
}

Expr <- d:Decimal ExprTail { push(' ', d), eval() }
ExprTail <- "+" d:Decimal ExprTail { push('+', d), 0 }
        /   "-" d:Decimal ExprTail { push('-', d), 0 }
        /   % { 0 }

Decimal <- "-"? [0-9]+ { atoi(ref_str()) }
```

14 Interactive inputs

Leaving aside problematic left association, let's go back to our previous calculator. You'll recall that this handles addition and multiplication, with arbitrary whitespace between numbers and symbols. See Chapter 12 [Handling whitespace], page 12.

Suppose we wanted to make an interactive calculator, with a read-eval-print loop. It should print a prompt, read a line from the user, evaluate it, print the result, and then print another prompt. That's easy enough to achieve, as each line read from the user will be a complete utterance in our grammar.

That interface would doubtless be fine in practice for a simple calculator. But for a more complex interactive language, we might like a more sophisticated read-eval-print loop. To demonstrate this, let's artificially complicate the calculator. Suppose we terminate each sum with an equals character, so a typical input would be '`2+3=`'. Now that we can recognise the end of an utterance, we can allow the user to enter an expression that spreads over several lines.

For a calculator, this is overkill, but it's exactly how command-line interfaces such as the Unix shells, SQL front-ends, and Lisp-like languages typically work. Normally, a different prompt is printed when further input is required; the classic example would be the '`PS1`' and '`PS2`' prompts in the Bourne shell. Pacc has support for building interfaces like these.

First, we need to mark in our grammar the points where the input can be split across multiple lines. We do this with the symbol '`$`'. This needs to go in high level rules, in all the places where it makes sense for the user to hit `RET` and keep typing, but not, for instance, in the middle of a number.

To strip the idea right down to its basics, consider this trivial grammar, which sums two digits, possibly on separate lines. Note that each digit is optionally followed by '`\n`', so valid utterances include both unsplit inputs, such as '`23\n`', and inputs split onto two lines, such as '`2\n3\n`'.

```
Sum :: int <- a:Digit $ b:Digit { a + b }
Digit <- [0-9] "\n"? { ref_0(ref()) - '0' }
```

There is only one possible place to break the input: between the two digits. We have marked that point in the grammar with the '`$`'. Now we can invoke pacc with the '`-f`' flag, and it will produce *two* parsers. The first parser is a perfectly normal one; the '`$`' has no effect at all.

For the second parser, pacc modifies the grammar to something like this:

```
Sum :: void <- Digit Digit?
Digit <- [0-9] "\n"?
```

The second digit (everything after the '`$`') is now optional. Additionally, semantic expressions have been removed: this grammar is for recognising only, it can't evaluate anything.

We can use the two grammars to construct an interactive two-digit-summer as follows.

1. Print the normal prompt, and read some input from the user.
2. Hand that input to the normal parser.
3. If the normal parser succeeds, print the result, and repeat from step 1.
4. Otherwise, hand the input to the second parser.
5. If the second parser succeeds, print the secondary prompt, read some more input from the user, append it to the input so far, and repeat from step 2.
6. Otherwise, report the parsing error, and repeat from step 1.

Here's what the complete grammar for the interactive calculator will look like, `icalc.pacc`. As well as the '`$`' markers, we have added the '`Equals`' rule that indicates the end of an expression.

```
Expression <- _ s:Sum $ Equals -> s
```

```
Sum <- p:Product $ Plus $ s:Sum -> { p + s }
     / Product
Product <- t:Term $ Star $ p:Product -> { t * p }
         / Term
Term <- Decimal / lBrace $ s:Sum $ rBrace -> s
# lexical elements
Decimal <- [0-9]+ _ -> { atoi(ref_str()) }
Plus :: void <- "+" _
Star <- "*" _
Equals <- "=" _
lBrace <- "(" _
rBrace <- ")" _
_ <- [ \t\n]*
```

Here's a wrapper program, `main.c`.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "parse.h"

#define LINE 80
#define P1 "$ "
#define P2 "> "

int main(void) {
    char *text = 0;
    char *prompt = P1;
    char line[LINE + 1];
    int len, parsed;
    struct pacc_parser *p0 = pacc_new();
    struct pacc_parser *p1 = pacc_feed_new();

    text = malloc(1); if (!text) exit(1);
    len = 0; *text = '\0'; prompt = P1;
    for (;;) {
        printf("%s", prompt); fflush(stdout);
        if (!fgets(line, LINE, stdin)) break;
        len += strlen(line);
        text = realloc(text, len);
        if (!text) exit(1);
        strcat(text, line);

        pacc_input(p0, text, len);
        parsed = pacc_parse(p0);
        if (parsed) {
            printf("%d\n", pacc_result(p0));
            len = 0; *text = '\0'; prompt = P1;
        } else {
            pacc_feed_input(p1, text, len);
            parsed = pacc_feed_parse(p1);
            if (parsed) {
```

```
                prompt = P2;
            } else {
                char *e = pacc_feed_error(p1);
                fprintf(stderr, "%s\n", e);
                free(e);
                len = 0; *text = '\0'; prompt = P1;
            }
        }
    }
    if (len) fprintf(stderr, "unexpected end of input\n");
    else fprintf(stderr, "\n");
    return 0;
}
```

And, for completeness, here's a `Makefile`. (This example can be found in the `test/icalc/` directory of the pacc source distribution.)

```
CFLAGS = -std=c99
PACC = ../../pacc

icalc: parse1.o parse2.o main.o
        $(CC) -o $@ $^

main.o: main.c parse.h
parse1.o: parse1.c
parse2.o: parse2.c

parse.h parse1.c parse2.c: icalc.pacc
        $(PACC) -dparse.h -o parse1.c -f parse2.c $<

clean:
        rm -f icalc main.o parse1.o parse2.o parse.h parse1.c parse2.c
```

15 The any matcher

There is one matcher that we have not yet met. It is the *any* matcher, written as '`.`', a period. As you might expect, it matches a single (UTF8) character (including the newline character), without regard to what that character actually is. If you are familiar with regular expressions, using '`.`' to match any character may seem familiar, but beware! Pacc behaves quite differently from regular expression matchers.

15.1 Pacc expressions and regular expressions

For one thing, in regular expressions '`.`' does not (usually) match the newline character. Pacc is more uniform: '`.`' always matches *any* single character. But this is a minor detail. What I really want to demonstrate is how pacc's matching rules are different from those of regular expressions.

Let's look at an example. Consider the regular expression '`/.*X/`'. Under regular expression rules, '`*`' is said to be *greedy*, which means that '`.*`' matches as much as possible, *while allowing the rest of the expression to match.* Suppose the input string is '`fooXbarXbaz`'. In this case, the expression will match '`fooXbarX`'. In general, it will match to the last '`X`' in the input, or fail if there is no '`X`'.

The pacc expression '`.* "X"`' looks superficially similar, but in this case '`.*`' will match *the entire string*, and there is nothing left for the '`"X"`' to match. So '`.* "X"`' will always fail. (If '`*`' is greedy in regular expressions, it's positively gluttonous in pacc!)

The matching engine for regular expressions effectively shifts input characters back and forth between different parts of the regular expression (such as the '`.*`' and the '`X`' in our example), with the aim of always finding a match if it possibly can. This is sometimes called *backtracking*.

Pacc never backtracks. Instead, each matcher in a pacc expression is considered in turn, and matches as much as possible. If that causes an overall failure, pacc simply moves on to the next alternative. So pacc expressions must be carefully written so they cannot match more than they should.

In this case, we can use a negated character class matcher instead of the any matcher. The pacc expression '`[^X]* "X"`' is closer in meaning to the regular expression we started with. Of course, this will only match '`fooX`'. (It's actually a common mistake when using regular expressions to write '`/.*X/`' intending that it will only match to the first '`X`'. When that is what is required, we must adopt the same technique as the pacc expression and write '`/[^X]*X/`', or resort to Perl's non-greedy '`*?`' operator if it's available: '`/.*?X/`'.)

To match up to the *last* '`X`' in pacc, we simply need to apply a repetition operator to the previous expression: '`([^X]* "X")+`'. Apart from newline handling, this is exactly equivalent to the regular expression we started with: it matches to the last '`X`' in the input, and fails if there is no '`X`'.

16 Guards

In the last section, we introduced the any matcher '`.`' which matches any character all, and then decided instead to use a negated character class matcher which matches *almost* any character. To make much use of the any matcher, we need guards.

A matcher really does *two* jobs. First, it *checks* to see if the input at the current position matches. Then, if it does, it *consumes* that input by moving the current position forwards.

A *guard* is a matcher modified so that it only checks, and never consumes. Any matcher can be turned into a guard by prefixing it with '`&`'. For example, the rule '`Proto <- ( "ftp" / "http" ) &":"`' matches either of two well-known protocols, but *only* if it is immediately followed by a colon '`:`'.

Guards made with the '`&`' are *positive* guards. They are not actually as useful as *negative* guards; any matcher can be turned into a negative guard by prefixing it with '`!`'. A negative guard succeeds, provided the input at the current position does *not* match. Negative guards work well in conjunction with the any matcher.

The venerable language Pascal surrounds comments with braces: '`{`' and '`}`'. (Pascal comments do not nest: '`{`' may not occur within a comment.) We can easily write a pacc rule to match Pascal comments, using a negated character class again:

```
PascalComment <- "{" [^{}]* "}"
```

How would we match C comments? They are surrounded by '`/*`' and '`*/`'. (C comments also do not nest: '`/*`' is permitted within a comment, but it doesn't have any meaning.) Character classes aren't going to help here, as they deal with just one character at a time. We need a combination of a negative guard and the any matcher:

```
CComment <- "/*" ( !"*/" . )* "*/"
```

The key to this rule is the parenthesized expression: '`!"*/" .`'. This means "check that the input at the current position is not '`*/`', and then match any character at all". Then we apply the repetition operator '`*`' to that, and we get an expression that consumes everything up to, but not including, the next occurrence of '`*/`' in the input.

17 Semantic guards

A third type of guard is available in pacc: the *semantic guard*. It is written '`&{ e }`', where *e* is an arbitrary C expression of integer type. The expression *e* is evaluated whilst parsing, with variables bound in the current sequence and to the left of the guard in scope. A semantic guard succeeds if the expression evaluates true (i.e. non-zero), and fails if the expression is false.

For example, suppose you want to match an alphabetic character in a locale-aware way. Pacc doesn't know anything about locales, but you can use the standard `isalpha()` predicate in a semantic guard:

```
Alpha <- c:. &{ isalpha(c) }
```

See Chapter 20 [Memory management], page 23.

18 Bound literals

Many programming languages use *reserved words* which resemble identifiers lexically, but form part of the syntax of the language. Pacc's *bound literals* can help to express these. A bound literal is written as '`"string":Rule`'. It calls '`Rule`', which must have type '`char *`' or '`ref_t`', and succeeds only if '`Rule`' returns a value equal to '`"string"`'.

Here's a tiny language with reserved words.

```
S <- "add":Word a:Digit b:Digit { a + b }
   / "sub":Word a:Digit b:Digit { a - b }

Digit <- d:[0-9] " "* { ref_0(d) - '0' }

Word :: ref_t <- w:[a-z]+ " "+ -> w
```

19 Error handling

In the event that the parser returns '`0`' to indicate that the parse has failed, then you can call `char *pacc_error(struct pacc_parser *)` which returns a string indicating the furthest point in the input which the parser reached, and what it expected to find there.

Consider the following one-rule grammar:

```
FortySomething <- "forty-" ("four" ↦ { 44 } / "five" ↦ { 45 })
```

With the input '`fifty-three`', the parser can make no progress at all, and `pacc_error()` returns '`1:1: expected FortySomething`'.

With the input '`forty-three`', the first 6 characters are successfully parsed, and `pacc_error()` returns '`1:7: expected "four", or "five"`'.

Note that `pacc_error()` is the default name for the error function. If you used the '`--name=NAME`' / '`-n NAME`' flag, the error function will be called `NAME_error` instead.

There are currently some limitations on error handling.

- Only one error can ever be reported. There is no option for *error recovery*. For example, a parser for a language like C on encountering an error should ideally discard input till the next '`;`' or '`}`', and then resume parsing and reporting further errors it may find. This cannot be achieved with a pacc-created parser yet.
- Certain pacc constructs are *syntactic sugar*, and in the process of desugaring them, new rules are created. Currently errors may reference these names, which are meaningless to the end user. For example, if the above grammar is altered to the following by adding a '`*`' after the first string:

  ```
  FortySomething ← "forty-"* ("four" ↦ { 44 } / "five" ↦ { 45 })
  ```

  then the input '`forty-three`' produces the error '`1:7: expected FortySomething:*:0, "four", or "five"`'

20 Memory management

The parsing engine generated by pacc allocates a fair amount of memory. (To parse `pacc.pacc` itself on a 64-bit architecture, pacc allocates around 3.5MB of memory.) If a memory allocation call fails, the parsing engine writes the message '`pacc: panic: out of memory`' to `stderr` and exits non-zero.

When you call `pacc_destroy()`, all memory allocated by the parsing engine itself is freed.

For any real application, semantic expressions will need to allocate memory to build an AST. Most of the time, pacc only calls the semantic expressions it needs, once the parse is complete. As long as you correctly free the AST you should be fine.

Semantic guards are problematic, though: they are called, along with any expressions needed for variable bindings, during the parse. This may well result in memory being allocated that may not be referenced by the AST that is finally built. Currently there is no way to free such memory. This is a hard problem to solve, since the results calculated while evaluating semantic guards will be reused for the final result if they are needed.

The parsing engine written by pacc uses only `realloc()` to allocate memory, and `free()` to free memory. We avoid `malloc()` by observing that `malloc(x)` is equivalent to `realloc(0, x)`. The intention is to make it slightly easier to supply your own memory allocator: it need only implement the `realloc()` and `free()` interfaces (although of course it must be prepared to accept '`0`' as the first argument to `realloc()`).

For example, if you want the parsing engine to use the Boehm garbage collector, include the following in the preamble of your grammar.

```
#include <gc.h>
#define realloc(x,s) (GC_REALLOC((x),(s)))
#define free(x) ((void)x)
```

21 User manual

This *User Manual* is a complete but concise description of all the features of pacc.

21.1 Invoking pacc

Usage:

```
pacc [OPTION]... FILE
```

`pacc` must be invoked with the name of a grammar file, which conventionally has a `.pacc` extension. `pacc` will write an output file with the same name but a `.c` extension.

Option summary:

```
Operation modes
  -h, --help               display this help and exit
  -v, --version            report version number and exit
  -D, --dump-ast=WHEN      dump the parse tree at various points

Parser:
  -n, --name=NAME          name the grammar (default: pacc)
  -f, --feed=FILE          write extra feed parser to FILE

Output:
  -d, --defines[=FILE]     also produce a header file
  -o, --output=FILE        write output to FILE
```

21.1.1 Output options

Some options control the output of pacc.

'`--output=FILE`'
: With the option '`-oFILE`' or '`--output=FILE`', pacc will write its output to the named *FILE*. Since pacc's output is C source code, you should specify a *FILE* that has a `.c` extension.

'`--defines[=FILE]`'
: With the option '`-d`' or '`--defines`', pacc will additionally write a header file, containing external definitions. If no *FILE* is specified, the defines file will have the same name as the output file but with a `.h` extension.

21.1.2 Parser options

Some options control the parser (or parsers) that pacc generates.

'`--name=NAME`'
: The parser created by pacc is interfaced to your C program through a number of functions with names such as '`pacc_new()`' (see Section 21.7 [Interfaces], page 28). When the '`-nNAME`' or '`--name=NAME`' option is specified, the functions are instead named '*NAME*`_new()`'.

'`--feed=FILE`'
: With the option '`-fFILE`' or '`--feed=file`', pacc writes an extra *feed parser* to the named *FILE*. This can be used for parsing command line input where some lines are correct but incomplete (see Section 21.6 [Feeding], page 28). If the main parser returns an error, but the feed parser recognises the input, the controlling program should solicit more input from the user, and feed it to the parser.

21.1.3 Help options

The usual help options are supported.

'`--help`' If the '`-h`' or '`--help`' option is specified, pacc writes a short description of its command line usage and options to standard output, then exits.

'`--version`'
If the '`-v`' or '`--version`' option is specified, pacc writes version information and its copyright statement to standard output, then exits.

21.1.4 Debug options

Some options can help with debugging pacc itself.

'`--dump-ast=`*WHEN*'
If the '`-D`*WHEN*' or '`--dump-ast=`*WHEN*' option is specified, pacc will write a dump of the Abstract Syntax Tree representation. The *WHEN* string specifies various points at which the AST will be dumped: if it contains the character '`0`' the tree will be dumped as soon as the input grammar has been parsed; '`1`' dumps the tree after desugaring; '`2`' dumps the tree after *cooking* (preparing the grammar for feeding with the '`-f`' flag). Multiple dumps can be specified, e.g. '`-D02`'.

21.2 Grammar

The overall structure of a *pacc grammar* consists of an optional preamble enclosed in braces: '`{}`' followed by one or more rules. The *preamble* is C source code, and is copied verbatim near the beginning of the output file. (If you use the '`--defines`' option, the preamble is also copied into the `.h` file that it creates.)

21.2.1 Rules

A *rule* consists of a name, an optional type, and a definition. The first rule is the *start rule* of the grammar: valid utterances of the language defined by the grammar match the start rule.

The *name* follows the same lexical rules as C identifier names: it must start with an alphabetic character or underscore, and remaining characters are alphabetic, numeric, or underscores. The namespace of pacc rule names is entirely separate from any C program, however, and there are no reserved words.

The *type*, if present, is introduced by a double colon '`::`', and can be any C type name that is in scope, provided that name can be constructed with identifiers and the '`*`' character. (Function pointers can be used via a '`typedef`' in the preamble.) If the type is omitted, the rule has the same type as the previous rule. (**Caution**: this means that rearranging the rules in a grammar can change the meaning of the grammar!) If the first rule has no explicit type, its type is '`int`'.

The *definition* is introduced with a left arrow '`<-`', which is followed by one or more matchers, combined with operators.

21.2.2 Matchers

*Matcher*s are the fundamental building blocks of definitions.

Literal: '`"string"`'
A string enclosed in double quotes matches exactly that string. The usual C character escapes are supported, so '`\n`', '`\012`', '`\xA`', and '`\u000A`' all represent the ASCII line feed character.

Character class: '`[a-z0-9]`'
: A series of characters in square brackets matches any single one of those characters, which are written as in C, e.g. '`[\t\n]`' matches a tab or a newline. If the first character is caret '`^`', the class is negated; that is, it matches any single character except those in the class. A character range is expressed using two characters separated by a hyphen '`-`', and it matches any single character with a Unicode code point between those of the two characters inclusive, e.g. '`[a-z]`' matches any lower case English letter. The character to the left of the hyphen must be strictly less than the character to the right of the hyphen. To include a literal hyphen, make it the first or last character in the class. For a literal right square bracket, make it first. For a literal caret, make it anything other than the first character.

Call: '`rule_name`'
: A bare word names another rule—it is an error if there is no such rule defined in this pacc grammar. It matches anything that the named rule matches. Call matchers are not allowed in a left recursive position.

Any: '`.`'
: The period character matches any single character. (UTF-8 coding is assumed.)

21.2.3 Operators

Pacc has the following *operators*, which are shown with their precedence, in order from 5 to 0. In the descriptions that follow, '`e`' and '`f`' are arbitrary pacc expressions; '`c`' is an arbitrary C expression.

Parentheses: '`(e)`' (5)
: Parentheses are used for grouping.

Repetition: '`e?`' '`e*`' '`e+`' (4)
: '`e?`' matches '`e`' optionally (zero or one occurrences); '`e*`' matches zero or more repetitions of '`e`'; and '`e+`' matches one or more repetitions of '`e`'.

Guards: '`&e`' '`!e`' '`&{c}`' (3)
: '`&e`' requires that the input matches '`e`' at this point, but it does not consume any input. '`!e`' requires that the input does *not* match '`e`', and again does not consume anything. '`&{ ... }`' is a *semantic guard*: the braces hold an arbitrary C expression of type '`int`'. The expression is evaluated; the guard succeeds if its value is non-zero.

Binding: '`name:e`' (3)
: Matches whatever '`e`' matches, and binds the value of '`e`' to '`name`', which is then in scope till the end of this alternative, for use in semantic guards and expressions. See Section 21.4 [Binding], page 27.

Bound literal: '`"string":e`' (3)
: Matches if '`e`' matches and the value of '`e`' (which must be of type `char *` or `ref_t`) is equal to '`string`'.

Sequencing: '`e f`' (2)
: Matches '`e`' followed by '`f`'.

Empty: '`%`' (2)
: Epsilon '`%`' represents an empty sequence; it can always be omitted, but its use can improve readability.

Value: '`-> {c}`' (1)
: Defines the *value* of a sequence; '`c`' is an arbitrary C expression which must have the same type as the rule. If '`c`' is a name (C identifier rules) or a decimal integer, the braces are optional. Otherwise, the arrow is optional.

Alternation: '`e / f`' (0)
: If '`e`' matches, the alternation matches; otherwise, if '`f`' matches, the alternation matches. Priority is important: '`e`' is always tried before '`f`'.

21.3 Lexical details

21.3.1 Comments and whitespace

Whitespace between grammar elements is optional, except when needed to avoid ambiguity. A *comment* may appear anywhere that whitespace can. Pacc understands both styles of C comment: '`/* comment */`' and '`// to end of line`' and also '`# to end of line`'.

21.3.2 Alternative characters

(Currently, the Unicode characters in the next few paragraphs are not visible in this version of the documentation.)

The left arrow that separates a rule (and optional type) from its definition can be written '←' (U+2190 LEFTWARDS ARROW), or '`<-`' (ASCII less than, hyphen) or '`=`' (ASCII equals).

The right arrow that separates a sequence from a value can be written '↦' (U+2192 RIGHTWARDS ARROW), or '`->`' (ASCII hyphen, greater than).

The slash that separates alternatives can be written '`/`' (ASCII slash) or '`|`' (ASCII vertical bar).

The empty sequence can be represented with 'ϵ' (U+03B5 GREEK SMALL LETTER EPSILON) or '`%`' (ASCII percent). (It can also be simply omitted, although explicitly marking it aids readability.)

21.4 Binding

The binding operator is '`:`'. To its left is an identifier, following the usual C rules. To its right is any pacc expression, which must match the input at this point. The binding operator brings into scope the identifier on its left; this name remains in scope till the end of the sequence, that is, the end of the rule or enclosing parentheses, or the next '`/`' operator. This identifier may be used in expressions and semantic guards.

If the expression on the right of the '`:`' is a call matcher, then the type of the name bound is the same as the type of the rule called. In all other cases, the type of the name bound is '`ref_t`'; this includes the case where the expression is a parenthesized call.

21.5 References to the input

Expressions in a pacc grammar frequently need to refer to the input string. The type '`ref_t`' and a number of functions facilitate this.

'`ref_t`'
: An opaque type which can hold a reference to a substring of the input string. Whenever a name is bound to an expression that is not a simple rule call, it has the type '`ref_t`'. For example, in '`c:.`', the type of '`c`' is '`ref_t`'.

'`ref_t ref()`'
: Returns a reference to everything that is matched by the current rule.

'`char *ref_ptr(ref_t r)`'
: Returns a pointer to the start of the substring referenced by '`r`'.

'`char ref_0(ref_t r)`'
: Returns the first byte of the substring referenced by '`r`'.

‘`size_t ref_len(ref_t r)`’
: Returns the length of the substring referenced by ‘`r`’.

‘`char *ref_dup(ref_t r)`’
: Returns a newly-allocated, NUL-terminated copy of the substring. Almost equivalent to ‘`strndup(ref_ptr(r), ref_len(r))`’, except that it is more portable, and it calls ‘`nomem()`’ if memory cannot be allocated.

‘`char *ref_str()`’
: Equivalent to ‘`ref_dup(ref())`’.

‘`int ref_streq(ref_t r, char *s)`’
: Tests if the substring referenced by ‘`r`’ is equal to the NUL-terminatd string ‘`s`’. Has the same value as ‘`!strcmp(ref_dup(r), s)`’, but allocates no memory.

21.6 Feeding

Feeding is the mechanism which allows pacc to handle input coming from interactive streams. The user's input may be a complete utterance, or it may be the start of an utterance. In the latter case, we need to solicit more input.

The points in the grammar where the input may be broken are marked with the ‘`$`’ token.

Then pacc is invoked with the ‘`-f=FILE`’ option, it writes its normal output as usual, then it writes an *additional* parser to the specified ‘`FILE`’ for a grammar which has been modified in the following ways:

1. all semantic expressions have been removed;
2. wherever the ‘`$`’ token occurs in a sequence, the rest of that sequence is optional;
3. the interface functions are named ‘`pacc_feed_new()`’, etc. (or if ‘`--name=`*`NAME`*’ was specified, ‘*`NAME`*`_feed_new()`’, etc.).

21.7 Interfaces

The *external interface* specifies how your program invokes the pacc parser. The *internal interface* has considerations for the snippets of C code that appear in a ‘`.pacc`’ file, both in the preamble, and the value expressions of parsing rules.

21.7.1 External interface

For a parser with a return type of ‘`result`’, the external functions available (and defined in the header file written with the ‘`-d`’ flag) will look like this:

```
#define PACC_TYPE /* type of start rule */
struct pacc_parser;
int pacc_trace;
struct pacc_parser *pacc_new(void);
void pacc_input(struct pacc_parser *, char *input, size_t length);
void pacc_destroy(struct pacc_parser *);
int pacc_parse(struct pacc_parser *);
PACC_TYPE pacc_result(struct pacc_parser *);
char *pacc_error(struct pacc_parser *);
char *pacc_pos(struct pacc_parser *, const char *);
int pacc_wrap(const char *, char *, size_t, PACC_TYPE *result);
```

The ‘`pacc`’ prefix on each function name is the default. You can specify a different prefix with the ‘`--name`’ option. The structure will still be called ‘`pacc_parser`’ though.

The interface is an object-oriented one:

`struct pacc_parser *pacc_new(void)`
: returns a new parser object.

`void pacc_input(struct pacc_parser *p, char *input, size_t length)`
: prepares the parser to parse the given '`input`' of given '`length`'.

`int pacc_parse(struct pacc_parser *p)`
: performs the parse, returning '`1`' if it was successful, '`0`' otherwise.

`char *pacc_error(struct pacc_parser *p)`
: if `pacc_parse()` returned '`0`', returns a newly-allocated string describing the error that the parser found.

`char *pacc_pos(struct pacc_parser *p, const char *s)`
: returns a newly allocated string containing the coordinates of the furthest error found, e.g. '`3:17: `', followed by '`s`'.

`PACC_TYPE pacc_result(struct pacc_parser *p)`
: if `pacc_parse()` returned '`1`', returns the result of the parse.

`void pacc_destroy(pacc_parser *p)`
: destroys the parser.

`int pacc_wrap(const char *n, char *in, size_t l, PACC_TYPE *result_pointer)`
: is a convenience function that creates a parser and invokes it on the given input '`in`' of length '`l`'. If the input was successfully parsed, the result is evaluated into `result_pointer` and '`1`' is returned. Otherwise, the parse error is written to `stderr` prefixed by '`name`', and '`0`' is returned. In either case, the parser is destroyed.

21.7.2 Internal interface

There are some utility functions defined by the parser engine that you may use in the C code snippets in a '`.pacc`' file.

`int PACC_LINE`
`int PACC_COL`
: are a pair of macros that return `int`s, representing the line number and column of the first character matched by the current rule.

`void pacc_nomem(void)`
: prints the message '`pacc: panic: out of memory`' to `stderr`, and exits. The parser engine calls `pacc_nomem()` whenever a call to `realloc()` fails.

`void pacc_panic(const char *error)`
: prints '`pacc: panic: `' followed by `error` to `stderr` and exits.

Within the C code snippets in a '`.pacc`' file, you must avoid using names that are used by the parser engine itself. All names beginning with '`pacc_`', '`PACC_`' and '`_pacc_`' are reserved to the parser engine.

In addition, the name '`ref`', and all names beginning '`ref_`' are reserved.

Bug: currently the parser engine uses some other names, such as '`cur`' and '`_status`'.

All allocation of memory done by the pacc parser engine uses only `realloc()` and `free()`. Thus, an alternative memory manager can be used with '`#define realloc(x,s) ... #define free(x) ...`' in the preamble.

Appendix A GNU Free Documentation License

Version 1.3, 3 November 2008

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

   You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

   You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both

covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

   A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
   B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
   C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
   D. Preserve all the copyright notices of the Document.
   E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
   F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
   G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
   H. Include an unaltered copy of this License.
   I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its

Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with. . . Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Index

P

R

S

T

U

V

W