SPLPetitParserSlideshow

Building composable parsers with PetitParser

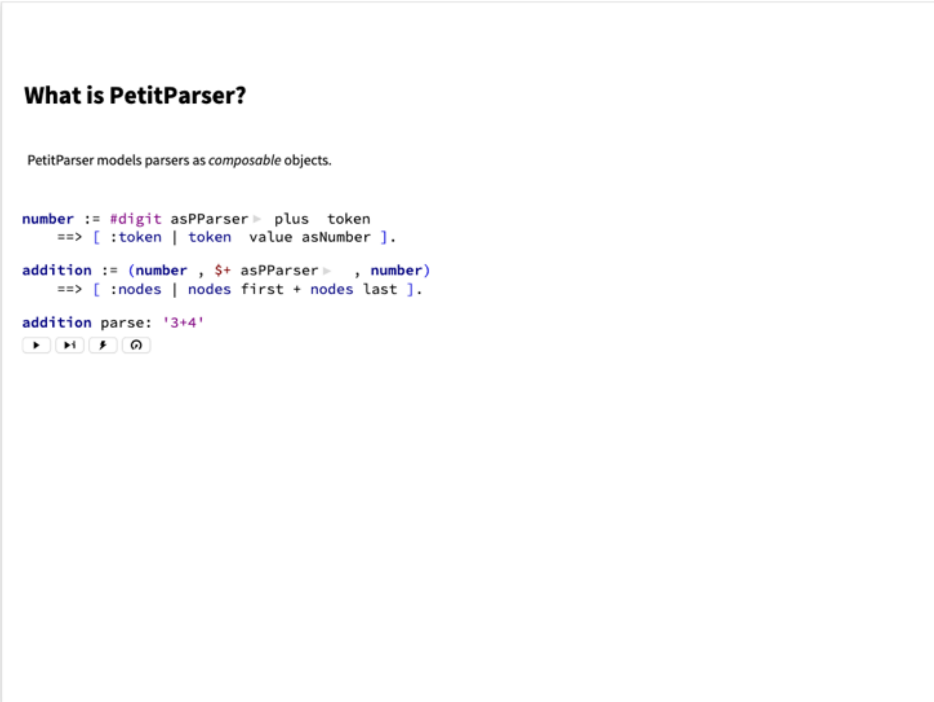
This slideshow is for the PetitParser lecture of the compiler construction course at UniBE.

What is PetitParser?

PetitParser is a parsing framework that combines several related parsing technologies: scannerless parsers, parser combinators, parsing expression grammars and packrat parsers.

The core idea is that parsers can be composed to form more complex parsers. That makes it convenient to develop and debug parsers.

In this example the `number` parser is composed of a `#digit` parser, and converts the parsed string to a number. The `addition` parser is composed of `number` and `+` parsers and performs the addition.



SPL Grammar

SPL is a simple, structured programming language with a compact grammar.

SPL Grammar

```
program        := declaration* EOF ;
declaration    := varDecl
               | statement ;
varDecl        := "var" IDENTIFIER ( "=" expression )? ";" ;
statement      := exprStmt
               | ifStmt
               | printStmt
               | whileStmt
               | block ;
exprStmt       := expression ";" ;
ifStmt         := "if" "(" expression ")" statement ( "else" statement )? ;
printStmt      := "print" expression ";" ;
whileStmt      := "while" "(" expression ")" statement ;
block          := "{" declaration* "}" ;
expression     := assignment ;
assignment     := IDENTIFIER "=" assignment
               | logic_or ;
logic_or       := logic_and ( "or" logic_and )* ;
logic_and      := equality ( "and" equality )* ;
equality       := comparison ( ( "!=" | "==" ) comparison )* ;
comparison     := term ( ( ">" | ">=" | "<" | "<=" ) term )* ;
term           := factor ( ( "-" | "+" ) factor )* ;
factor         := unary ( ( "/" | "*" ) unary )* ;
unary          := ( "!" | "-" ) unary
               | primary ;
primary        := "true" | "false" | NUMBER | STRING
               | "(" expression ")"
               | IDENTIFIER ;
```

An SPL example

SPL does not have procedures or objects, but it has loops, however, so it is still Turing-complete.

An SPL example

SPL is a minimal language with variables, expressions, `if`, `while` and `print` statements, but no procedures or classes.

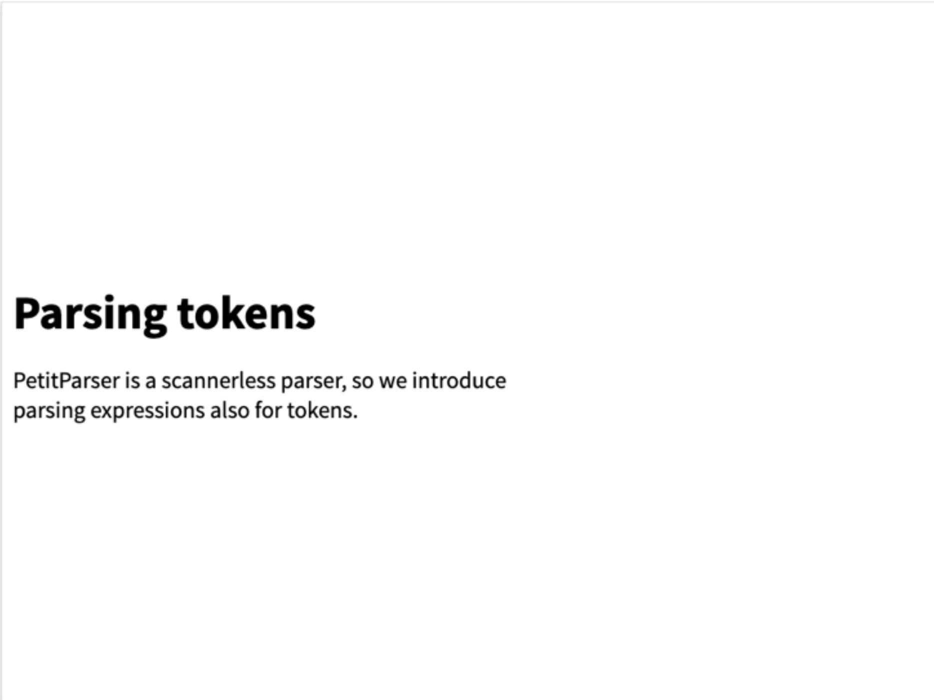
```
// Compute the factorial of arg
var arg=5;
var x=arg;
var fact=1;
while (x>0) {
    fact = fact * x;
    x = x - 1;
}
print fact;
```

Parsing tokens

PetitParser is a *scannerless* parser.

This means we do not have a separate scanner for individual tokens based on regular expressions, but instead we use parsing expressions for tokens as well.

We'll introduce parsing expressions for all of the SPL tokens, namely Boolens, integers, floats, strings, keywords and identifiers.

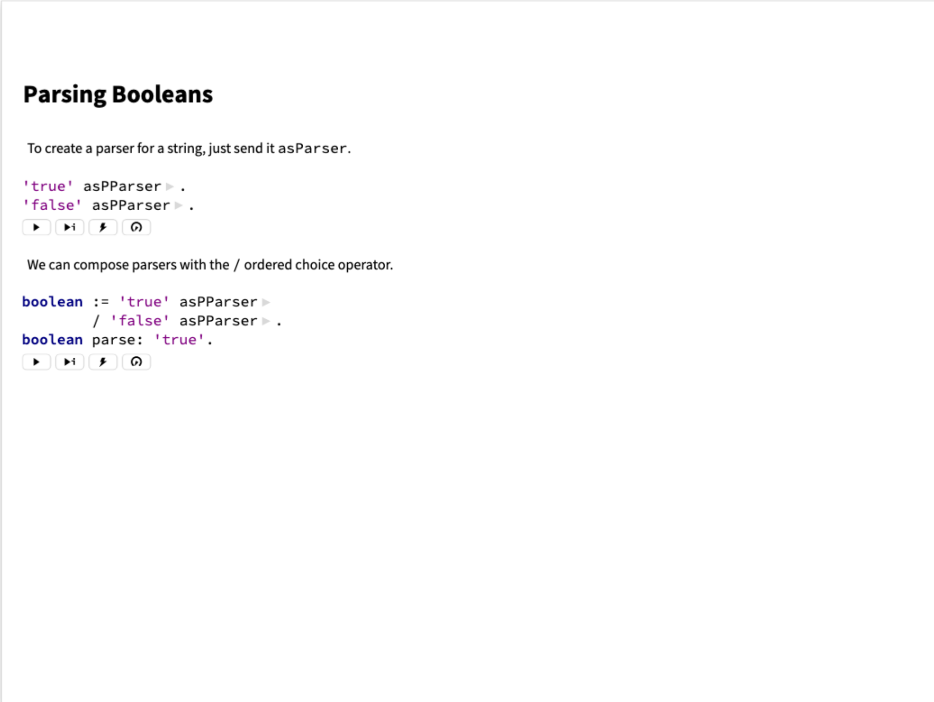


Parsing Booleans

To parse a character or a string, we just send it the message `asParser`.

Here we create two parsers, one which will parse the string `'true'`, and the other `'false'`.

We compose them with the *ordered choice* operator, `/`, to parse either `true` or `false`.



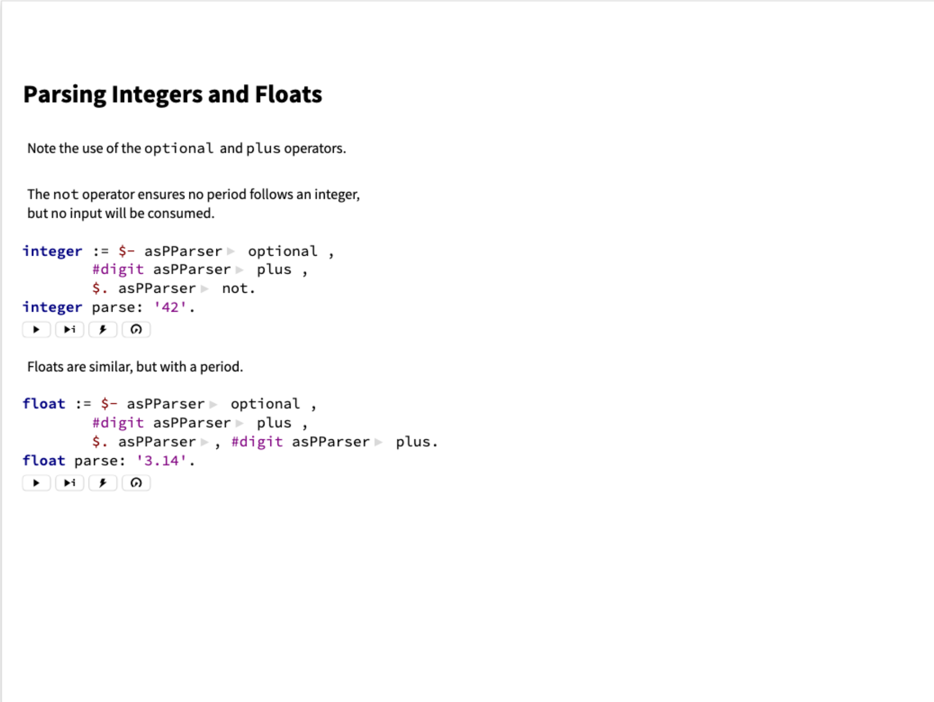
Parsing Integers and Floats

PetitParser makes use of numerous operators, or *combinators* to compose parsing expressions.

The `optional` operator creates a new parser that will parser either zero or one token. The `plus` operators will parse ` or more tokens.

The `not` operator does not consume a token, but simply fails if it sees the token, otherwise it succeeds. Here we make sure that an integer will only be recognized if there is no trailing period. If we add a period, the parse will fail.

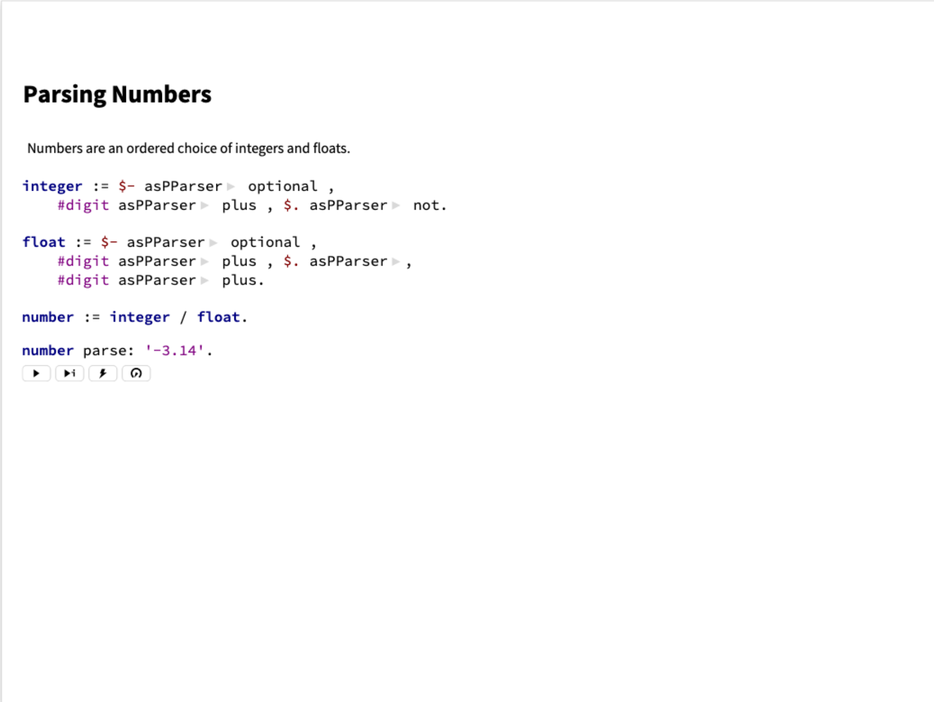
Parsing floats is similar, but in this case we *do* want the dot to be parsed.



Parsing Numbers

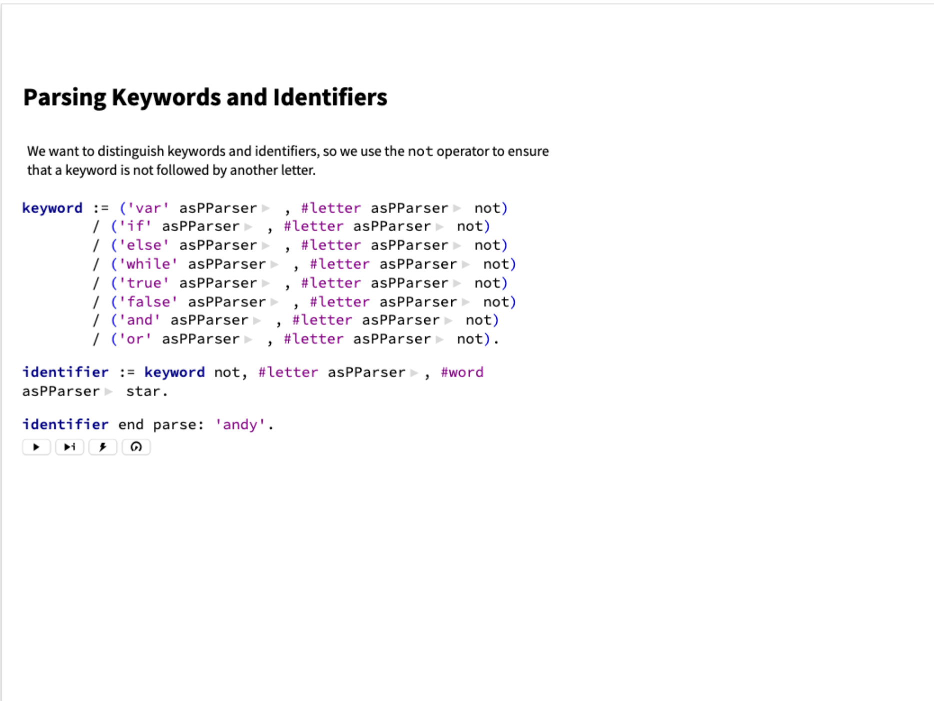
We can now combine the parsing expressions to recognize numbers as either integers or floats.

Note that the choice operator is strictly ordered. It will *first* attempt to parse an integer, and *only if that fails* will it try to parse a float.



Parsing Keywords and Identifiers

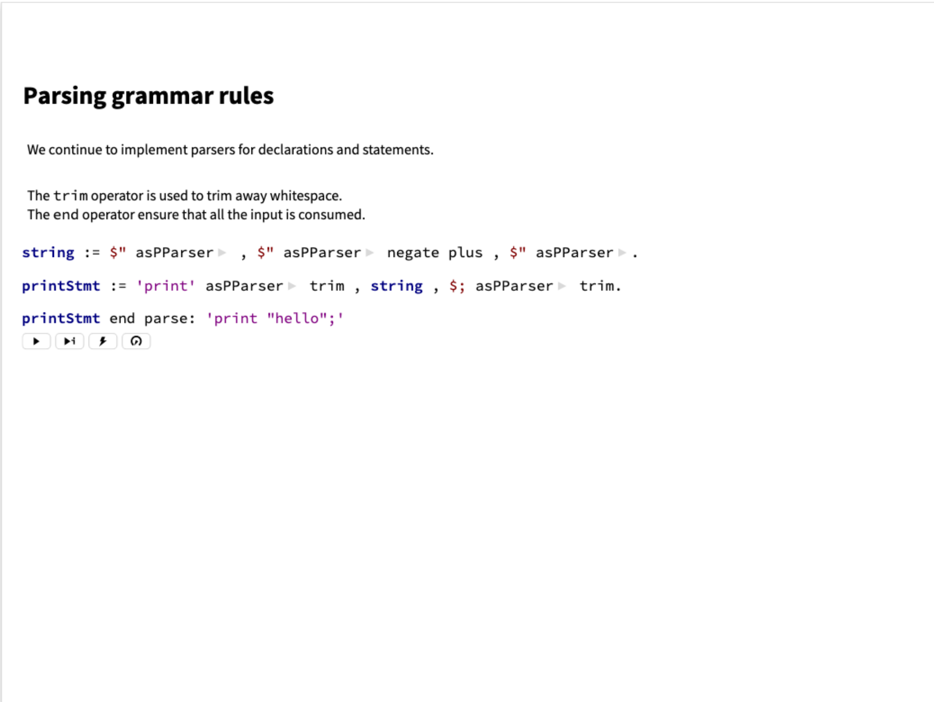
Here we use the `not` combinator to make sure we don't accidentally recognize the token `and` in identifiers such as `android` or `andy`.



Parsing grammar rules

Now we have parsing expressions for all the SPL tokens except strings. We can proceed to the actual grammar rules.

The `trim` operator makes it easy to get rid of whitespace following a token. The `end` operator will match the end of input, making sure that everything is parsed.

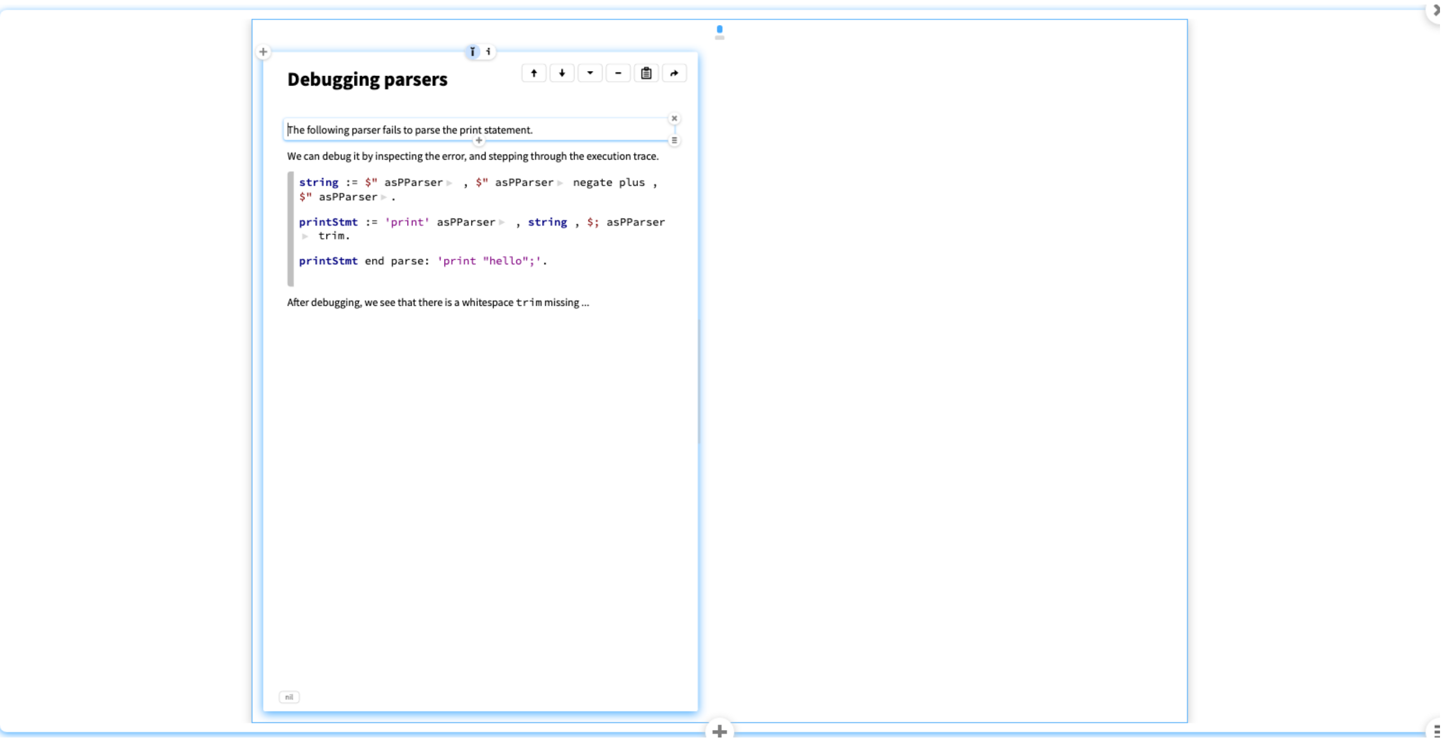


Debugging parsers

Here's a slightly buggy version of our `print` statement parser that fails with this input.

The result is a PetitParser “Failure” object that shows us the execution trace of the parser at the point where it failed. If we inspect this, we can walk through the tree to see how far it got.

We discover that after recognizing the “print” string, it expects a quotation mark for the start of a string, instead of whitespace. We fix this by adding the missing `trim` to the `print` parser.



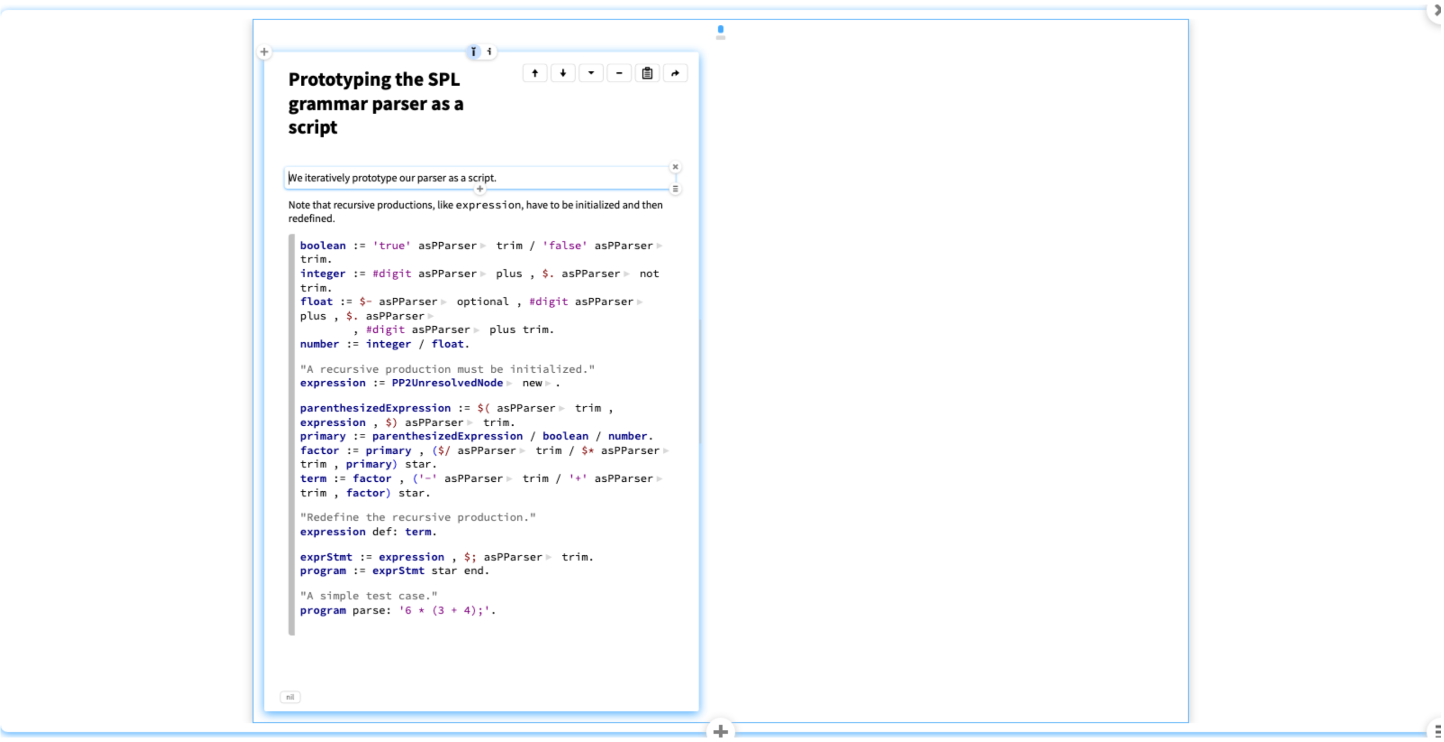
Prototyping the SPL grammar parser as a script

We continue to iteratively prototype the various SPL grammar rules until we have a complete grammar implemented as PetitParser parsing expressions.

Note that we must take special care with recursive grammar rules, as we cannot use parsers that have not yet been defined.

To break the recursion, the first time we introduce a recursive parser, such as `expression`, we define it as an instance of `PP2UnresolvedNode`. Then, once we have defined the other parsers that it needs for its own definition (and that use it recurseively, we *redefine* it use `def:`.

We can see that `expression` is later redefined as `term`.

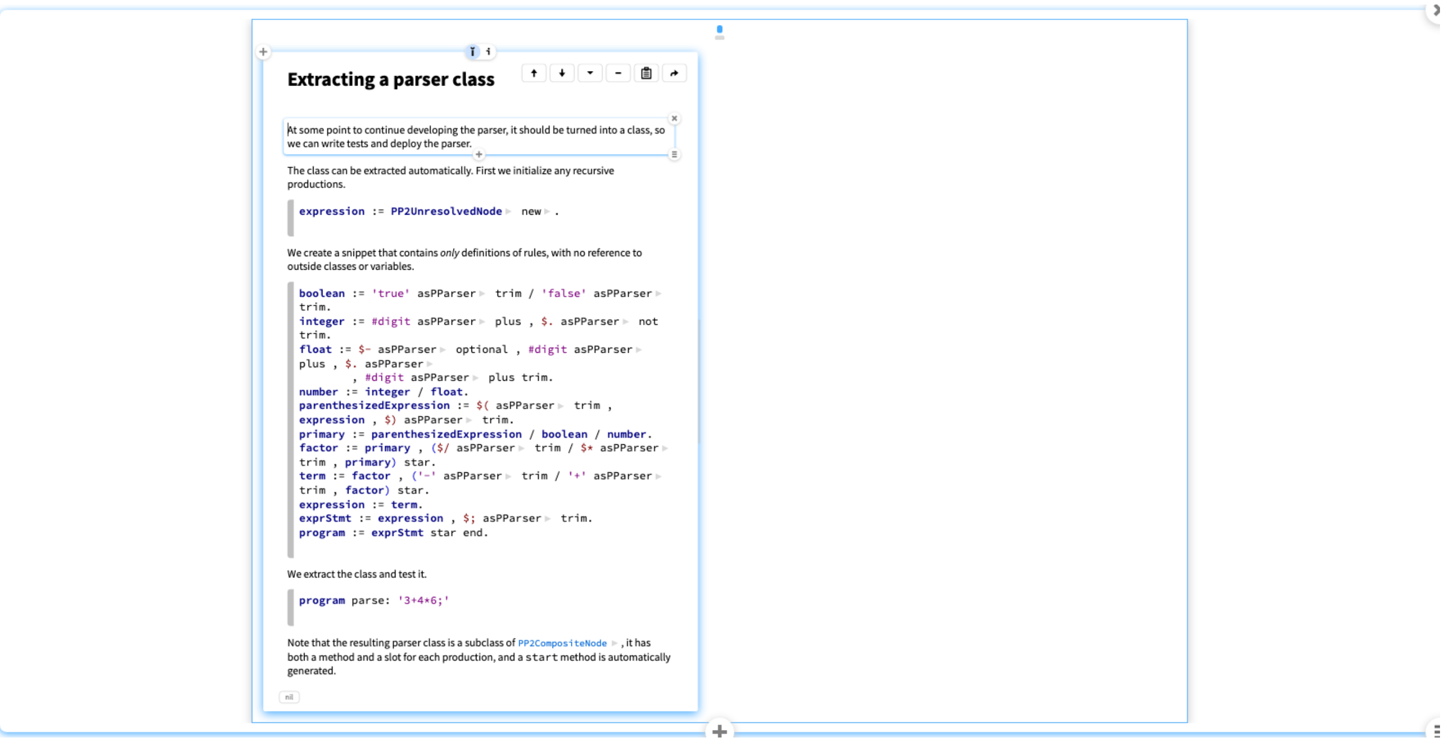


Extracting a parser class

Once we have a working script, we can apply a refactoring transformation to turn it into a class.

We initialize any recursive parsers, and then create a self-contained script that does not refer to any outside classe sor variables. We can right-click inside the script to `Extract PetitParser class`.

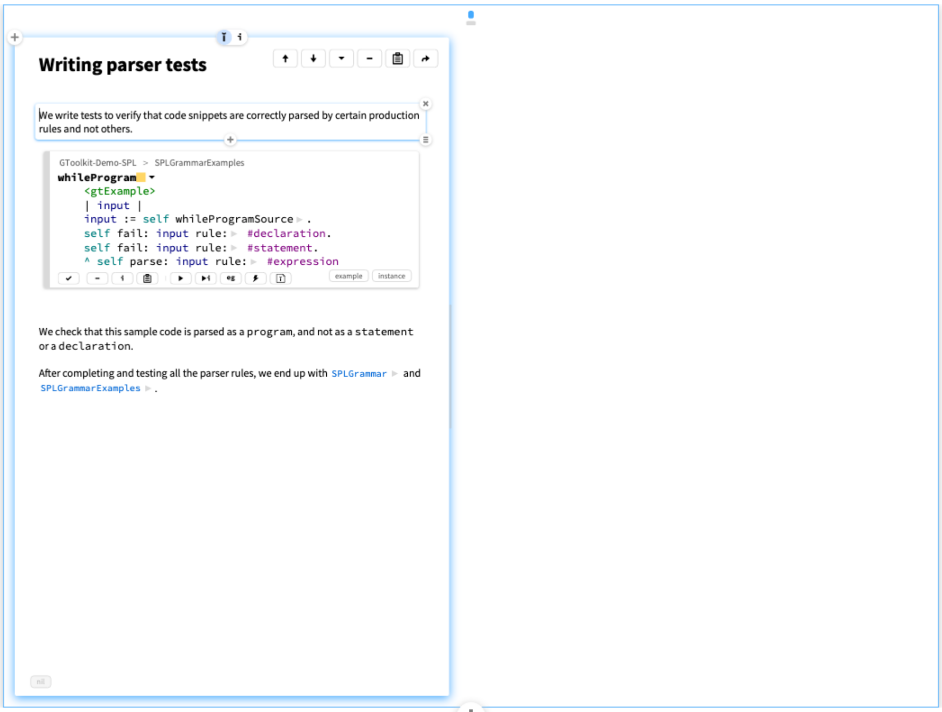
This creates a class in which each parsing expression is defined as a method, and its value is cached as an instance variable.



Writing parser tests

We write tests to ensure that each parsing expression and every grammar rule works as expected.

It's good practice to subclass `PP2CompositeNodeExamples`, a class that offers some utilities for testing parsers. In this example we simply test that a small program source can be parsed by the `program` parser, but not by `declaration` or `statement`.



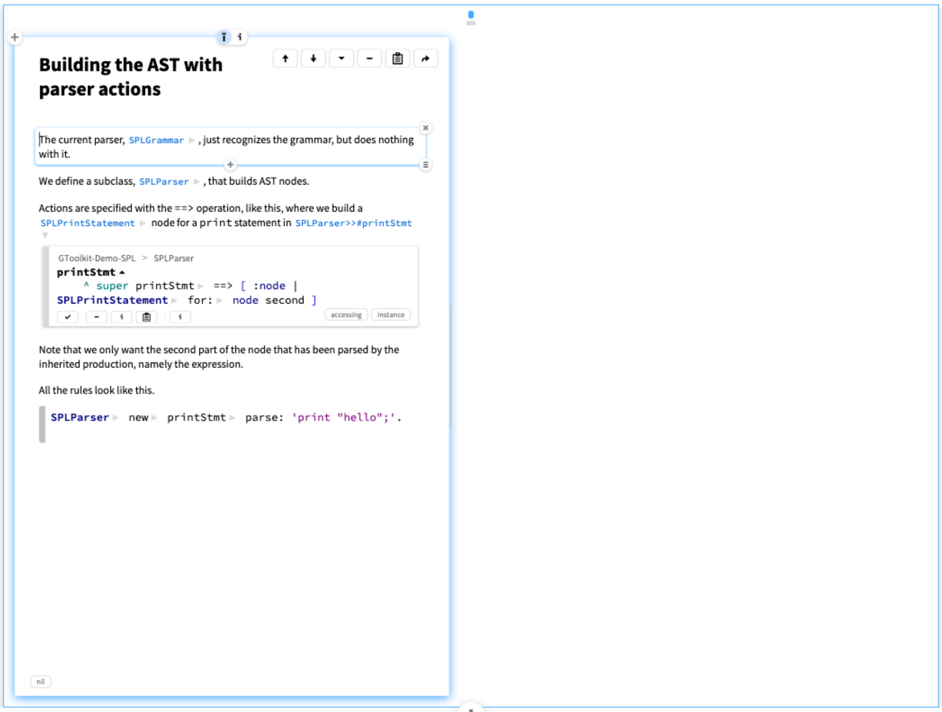
Building the AST with parser actions

The parser we have created just recognizes SPL programs, but doesn't perform any actions. Now we'd like to build an abstract syntax tree for the programs that are recognized.

In our very first example we saw that actions can be specified with the `==>` double arrow operator, which points to a block (*i.e.,* an anonymous function) that can transform the parsed data into something useful. We'll define a *subclass* of our basic grammar parser that decorates each parsing expression with an action. Note that the `SPLParser printStmt` method inherits `printStmt` from the superclass and adds the action.

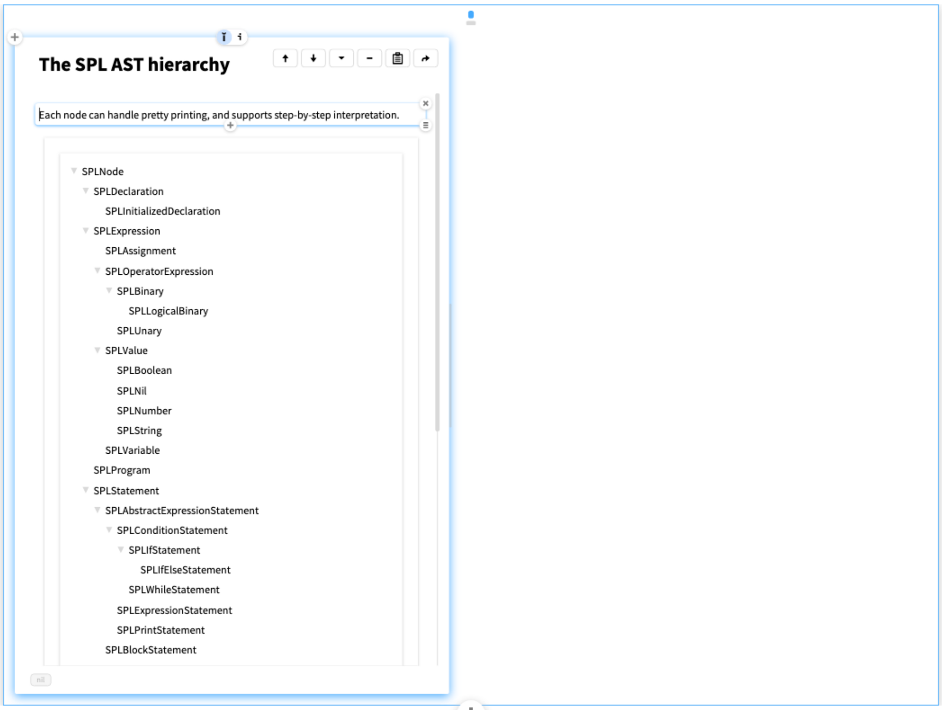
For each grammar rule we'll create an instance of an `SPLNode` subclass that will store the interesting bits in the instance variables of the AST node. In this case we create an `SPLPrintStatement` node and store the second part that was parse, namely the expression to be printed.

Now when we parse a bit of code with our refined parser expressions we get a proper AST node instead of an array of data.



The SPL AST hierarchy

We introduce an AST node class for every different syntactic element of the SPL grammar. Each leaf node can pretty-print itself, and can also perform an interpretation step.



SPL Semantics

We don't just want to parse SPL programs, but we also want to execute them.

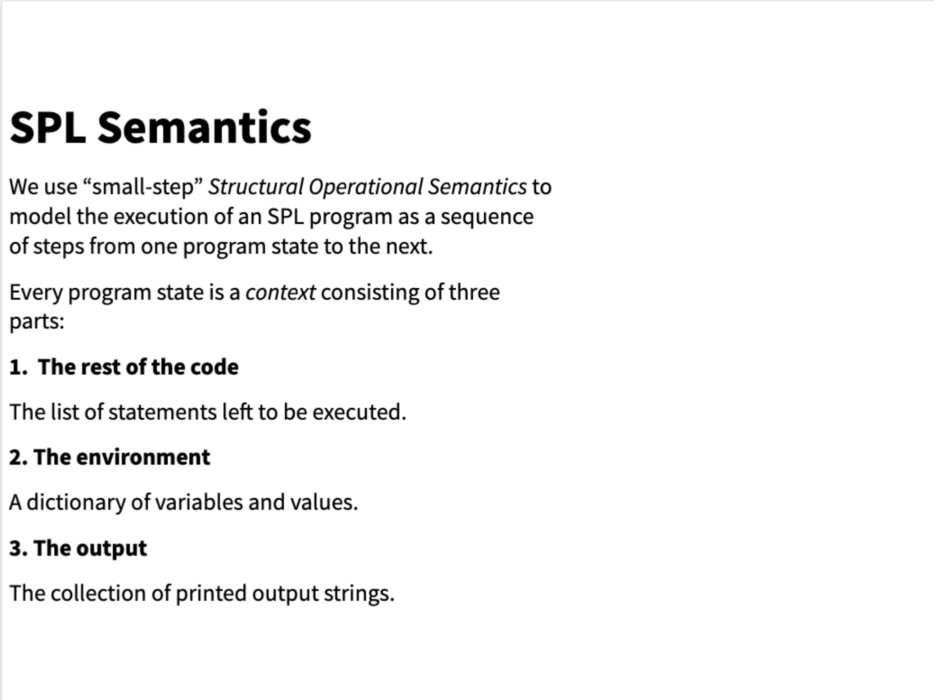
This can be done in numerous ways. We could generate bytecode for a virtual machine for an existing language, like Java, or we could directly compile SPL programs to machine code.

Another approach is to *interpret* SPL programs by transforming them, step-by-step, to simpler programs. This approach is called *Structural Operational Semantics* . We'd like to see each step of the execution, so we use what is called “small step” semantics.

SPL programs don't take any input except what is specified in the source code itself. Programs have variables, so we need to track the bindings of variables to values, and we need to track any output that is produced. That means that the *context* of a running SPL program consists in three parts: (1) the current “continuation”, *i.e.,* the “rest of the program” to be executed, (2) the environment of variables and their values, and (3) the output so far.

When we start executing, the continuation is the full program, the environment is empty, and so is the output.

When the program ends, the continuation is empty (the empty program), the environment contains the set of all variables and their final values, and the output is the final list of everything that has been printed.

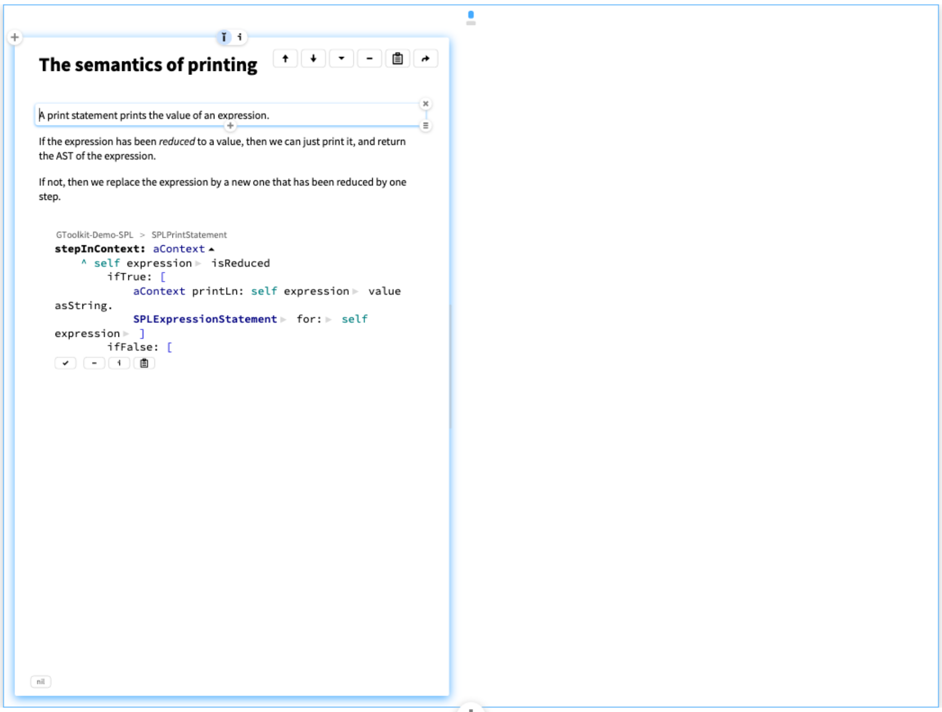


The semantics of printing

Every SPL AST node has a `stepInContext:` method that allows it perform one reduction step, and return a new, reduced AST.

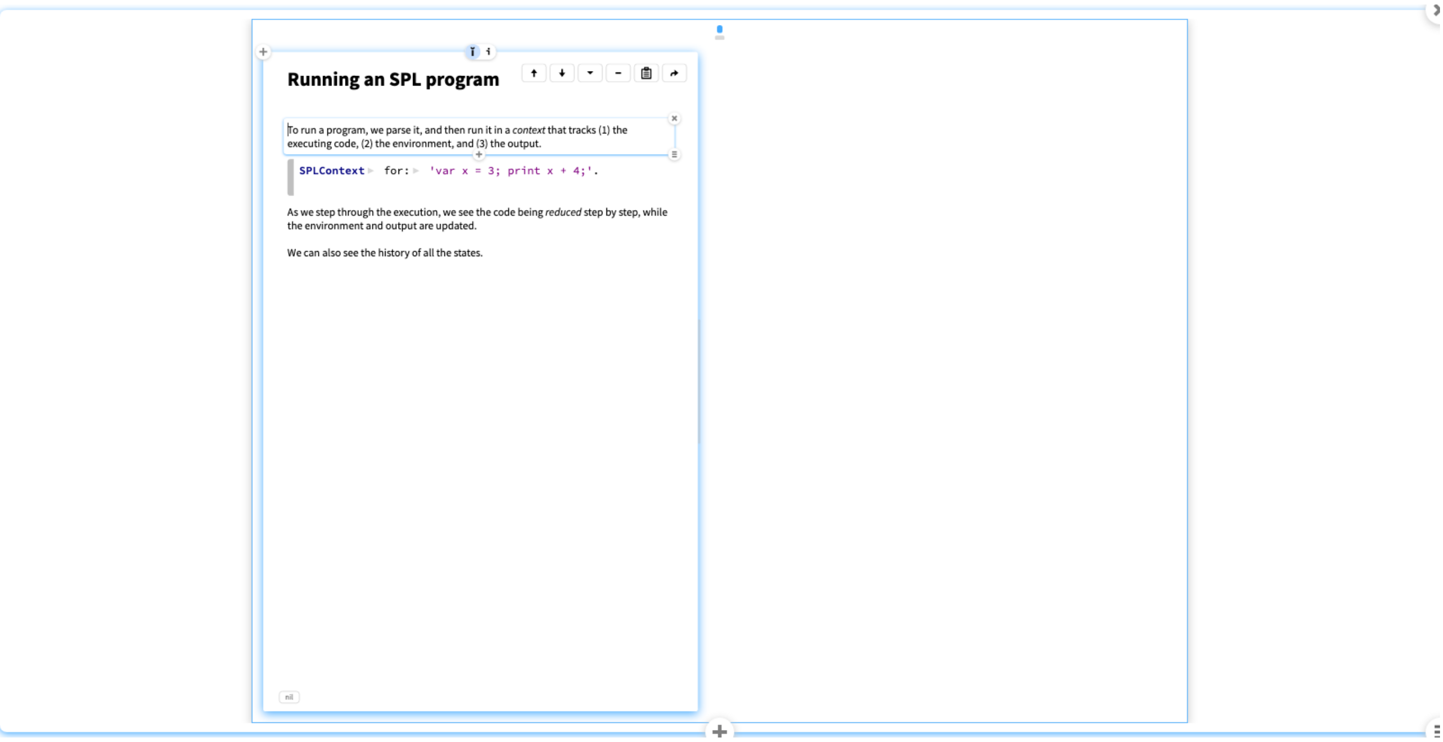
Let's just look at one of these, namely that of the `print` statement. A print statement prints the value of an expression. The reduction step, then, should check if the expressions value is already known, in which case we can just print it. If not, we have to perform a reduction step.

In the first case we return the AST for the reduced expression, which will be discarded in the next step, and in the second case, we return a new print statement AST with the expression redcued by one step. In either case we make some small progress.



Running an SPL program

To run an SPL program, we create a new context holding the AST of the program as its continuation, and an empty environment and output. If we inspect this object, we can then step through the execution, and also explore the history of all the reductions steps.



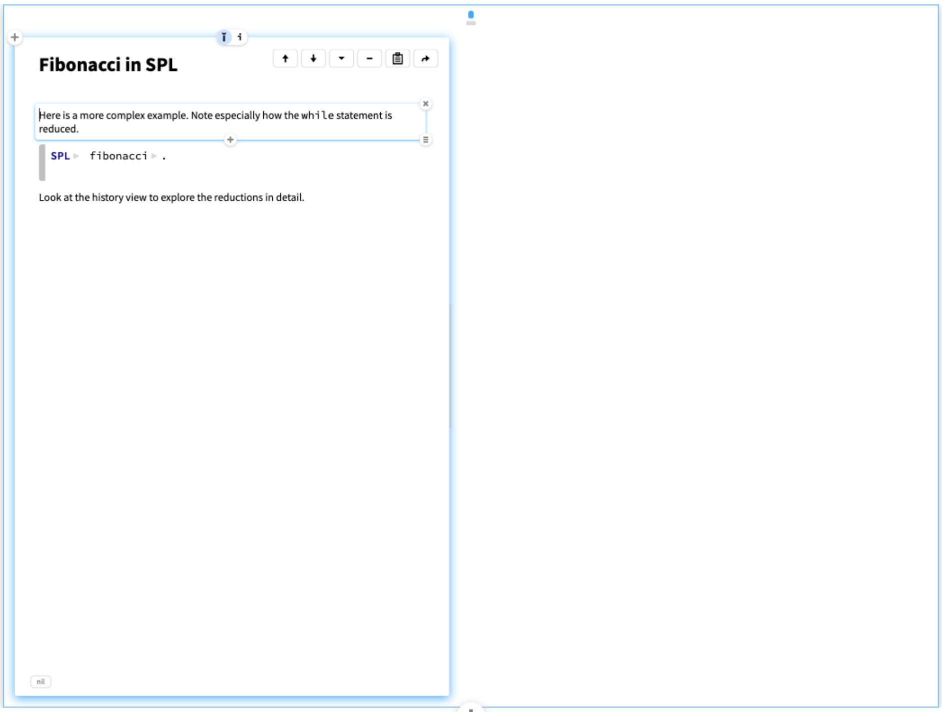
Fibonacci in SPL

Here's a Fibonacci program in SPL.

Since SPL doesn't have procedures, we cannot define a Fibonacci function, but we can make it into a program. Also SPL programs don't take arguments, so we have to encode the argument as a variable in the first line.

The algorithm simply keeps track of the last two Fibonacci values, printing out each new value computed, and terminating when we reach the required number of iterations.

Notice how the semantics of the `while` statement has been implemented by *unfolding* the while into an if statement where the “then” part contains one iteration of the body followed by another copy of the while loop.



Coda

You can explore the SPL case study for yourself by downloading Glamorous Toolkit from gtoolkit.com and going to the page “PetitParser SPL case study” in the GT Book.