Assignment 3 (main part): A Boa Parser

Version 1.0

Due: 22:00 on Friday, September 30, 2022

The objective of this assignment is to gain practical experience with using a parser-combinator library to construct a parser for a simple but realistic programming-language grammar. Please read through the *entire* assignment text before you start working on it.

Note This assignment also includes a collection of simple warm-up exercises, as described on Absalon. For those, you are *only* asked to submit your working code, but not a separate design/implementation document, assessment, or evidence of testing. If you need to communicate anything extra about your solutions to the warm-up exercises, place your remarks as comments in the source code.

1 The Boa concrete syntax

In Assignment 2, we introduced the *abstract* syntax of Boa, reproduced here for easy reference:

```
data Value =
    NoneVal
  | TrueVal | FalseVal
  | IntVal Int
  | StringVal String
  | ListVal [Value]
  deriving (Eq, Show, Read)

data Exp =
    Const Value
  | Var VName
  | Oper Op Exp Exp
  | Not Exp
  | Call FName [Exp]
  | List [Exp]
  | Compr Exp [CClause]
  deriving (Eq, Show, Read)

type VName = String
type FName = String

data Op = Plus | Minus | Times | Div | Mod | Eq | Less | Greater | In
  deriving (Eq, Show, Read)

data CClause =
    CCFor VName Exp
  | CCIf Exp
  deriving (Eq, Show, Read)
```

```
type Program = [Stmt]

data Stmt =
    SDef VName Exp
  | SExp Exp
  deriving (Eq, Show, Read)
```

Correspondingly, the *concrete* syntax of Boa is shown in Figure 1. In naming the nonterminals, we use the informal mnemonic convention that a *Foos* is a sequence containing at least one *Foo*, whereas a *Fooz* sequence may also be of length zero.

Supplementing the formal grammar, we also specify the following:

Lexical specifications of complex terminals There are three terminal symbols with nontrivial internal structure:

ident Identifiers (used for variable and function names) consist of one or more letters, digits, and underscores, where the first character must not be a digit. Additionally, an identifier must not be one of the Boa reserved words: `None`, `True`, `False`, `for`, `if`, `in`, and `not`. (For compatibility, it may be advisable to also avoid using other Python reserved words in Boa programs, but your parser should *not* treat those specially.)

numConst Numeric constants consist of an optional negative sign (`-`), followed (without any intervening whitespace) by one or more decimal digits. The first digit must not be a zero unless it is the *only* digit. Thus, “`-0`” or “`100`” are well formed *numConst*s, while “`007`”, “`+2`”, or “`- 4`” are not. Do not worry about `Int`-overflows.

stringConst String constants in Boa are written between single quotes (`'`). (Unlike in Python, neither the `"` nor `"""` delimiters are supported.) Inside a string constant, all *printable* ASCII characters are allowed, except for single quotes and backslashes, which must be escaped as `\'` and `\\`, respectively. Raw newline characters (being non-printable) are not allowed, but can be included by the sequence `\n`. On the other hand, to allow string constants to be written over multiple lines, a backslash followed immediately by a newline causes *both* characters to be ignored. Thus, the Boa string constant

```
'fo\\o\
b\na\'r'
```

should be parsed as the 9 characters “`fo\ob↵a'r`” (where ‘↵’ represents a single newline character). All other escape sequences (i.e., a backslash followed by anything other than a single quote, another backslash, the lowercase letter ‘n’, or a newline) are illegal.

General lexical conventions All identifiers and keywords are case sensitive.

Tokens of the grammar may be surrounded by arbitrary whitespace (spaces, tabs, and newlines). Some whitespace is *required* between identifiers/keywords and any immediately following letters, digits, or underscores. (No whitespace is needed *after* a numeric constant, unless it is immediately followed by another digit.)

Comments start with a hash character (`#`) and run until the end of the line. A comment counts as whitespace for the purpose of separating tokens, so that, e.g., the input

```
not#comment
cool
```

Program	::=	*Stmts*
Stmts	::=	*Stmt*
	|	*Stmt* ‘`;`’ *Stmts*
Stmt	::=	*ident* ‘`=`’ *Expr*
	|	*Expr*
Expr	::=	*numConst*
	|	*stringConst*
	|	‘`None`’ | ‘`True`’ | ‘`False`’
	|	*ident*
	|	*Expr* *Oper* *Expr*
	|	‘`not`’ *Expr*
	|	‘`(`’ *Expr* ‘`)`’
	|	*ident* ‘`(`’ *Exprz* ‘`)`’
	|	‘`[`’ *Exprz* ‘`]`’
	|	‘`[`’ *Expr* *ForClause* *Clausez* ‘`]`’
Oper	::=	‘`+`’ | ‘`-`’ | ‘`*`’ | ‘`//`’ | ‘`%`’
	|	‘`==`’ | ‘`!=`’ | ‘`<`’ | ‘`<=`’ | ‘`>`’ | ‘`>=`’
	|	‘`in`’ | ‘`not`’ ‘`in`’
ForClause	::=	‘`for`’ *ident* ‘`in`’ *Expr*
IfClause	::=	‘`if`’ *Expr*
Clausez	::=	ϵ
	|	*ForClause* *Clausez*
	|	*IfClause* *Clausez*
Exprz	::=	ϵ
	|	*Exprs*
Exprs	::=	*Expr*
	|	*Expr* ‘`,`’ *Exprs*
ident	::=	(see text)
numConst	::=	(see text)
stringConst	::=	(see text)

Figure 1: Concrete syntax of Boa

parses like `not cool`, not like `notcool`. Comments are *not* recognized within string constants, i.e., `#` behaves like any other character in that context.

Unlike Python, Boa is *not* indentation-sensitive. Thus, there might be Boa programs that would not be valid Python programs due to, e.g., leading whitespace on a line.

Disambiguation The raw grammar of *Expr* is highly ambiguous. To remedy this, we specify the following operator precedences, from loosest to tightest grouping:

1. The logical-negation operator '`not`'. Nesting is allowed, so `not not x < 3` parses like `not (not (x < 3))`.
2. All relational operators ('`==`', etc., including '`in`' and '`not`' '`in`'). These are all *non-associative*, i.e., chains like `x < y < z` are syntactically illegal (unlike in Python).
3. Additive arithmetic operators ('`+`' and '`-`'). These are *left-associative*, e.g., `x-y+z` parses like `(x-y)+z`.
4. Multiplicative arithmetic operators ('`*`', '`//`', and '`%`'). These are also left-associative.

Correspondence between concrete and abstract syntax This should be straightforward, with the following additional remarks:

- The relational operators '`!=`', '`<=`', '`>=`', and '`not`' '`in`' should be treated as syntactic sugar for the immediate negations of '`==`', '`>`', '`<`', and '`in`', respectively. For example, the input string "`x <= y`" should parse as the `Exp`-typed value `Not (Oper Greater (Var "x") (Var "y"))`.
- Numeric, string, and atomic constants should parse as the `Exp` constructor `Const` with a suitable `Value`-typed argument. For example, the Boa expression "`1 == True`" should parse as `Oper Eq (Const (IntVal 1)) (Const TrueVal)`, and the Boa string constant "`'a"b\n'`" as `Const (StringVal "a\"b\n")`.

  On the other hand, list constructors (penultimate alternative for *Expr* in the grammar) should always parse as `List [...]`, never `Const (ListVal [...])`, even if all the list elements are already constants.
- In the concrete syntax, a program must contain at least one statement, while in the abstract one, it is allowed to be completely empty. Likewise, the concrete syntax is more restrictive than the abstract one by specifying that a comprehension must always contain one or more clauses, starting with a `for`. Your *parser* should enforce these constraints, even though your *interpreter* was expected to also work without them.

2 The Boa parser

Your parser module code must be placed in the file `code/part2/src/BoaParser.hs`. (A stub implementation is provided.) You must use either the `ReadP` or the `Parsec` parser-combinator library (as supplied with the course-mandated LTS release). If you use `Parsec`, then only plain `Parsec` is allowed, namely the following submodules of `Text.Parsec`: `Prim`, `Char`, `Error`, `String`, and `Combinator` (or the compatibility modules in `Text.ParserCombinators.Parsec`); in particular you are *disallowed* to use `Text.Parsec.Token`, `Text.Parsec.Language`, and `Text.Parsec.Expr`.

Your parser module must implement (or import) a type `ParseError`, which must be an instance of (at least) the classes `Eq` and `Show`. It should export only a single function:

```
parseString :: String -> Either ParseError Program
```

When applied to a string representing a Boa program, this function should return either an AST for the program, or an error message that can be shown to the user. Don't worry about generating (more) informative error messages, beyond what your chosen parser library already provides.

In your report, you should document and explain any non-trivial modifications you did to the grammar to make it suitable for combinator-based parsing (notably how you dealt with ambiguity and/or left-recursion), as well as any other instances where you had to do something non-obvious, including in particular (if relevant): ***

- If using `ReadP`: explain where and *why* you used biased combinators (`<++`, `munch`) to suppress the normal full-backtracking behavior.
- If using `Parsec`: explain where and *why* you introduced backtracking/lookahead (`try`).

Driver program

We have upgraded the stand-alone main program `boa` so that it can now invoke both the interpreter and the parser. Thus, if you plug in your `BoaInterp` module from Assignment 2, you can now easily parse a Boa program (`-p` option), interpret an already parsed one (`-i`, like in the previous assignment), or both (the default), all from the command line.

Note that the actual file extensions are not significant for either Python or Boa, so that you should also be able to run most Boa programs with `python3 foo.boa`. Conversely, you can also try `boa foo.py`, as long as `foo.py` is restricted to the tiny Boa subset (and in particular, contains no function definitions or complex statements).

We will *not* (re)test your `BoaInterp` module for this assignment, but if you do include it with the submission (replacing the stubby one), please be sure that it does not break the build. As usual, `stack test` should build and run your parser tests.

[The rest of this document is essentially identical to that for Assignment 2, but is repeated here for easy reference. Any significant changes are highlighted in red.]

3 What to hand in

3.1 Code

Form To facilitate both human and automated feedback, it is very important that you closely follow the code-packaging instructions in this section. We provide skeleton/stub files for all the requested functionality in both the warm-up and the main part. These stub files are packaged in the handed-out `code.zip`. It contains a single directory `code/`, with a couple of subdirectories organized as Stack projects. You should edit the provided stub files as directed, and leave everything else unchanged.

It is crucial that you not change the provided types of any exported functions, as this will make your code incompatible with our testing framework. Also, *do not* remove the bindings for any functions you do not implement; just leave them as `undefined`.

When submitting the assignment, package your code up again as a single `code.zip` (*not* `.rar`, `.tar.gz`, or similar), with exactly the same structure as the original one. When rebuilding `code.zip`, please take care to include only the files that constitute your actual submission: your source code and supporting files (build configuration, tests, etc.), but *not* obsolete/experimental versions, backups, editor autosave files, revision-control metadata, `.stack-work` directories, and the like. If your final `code.zip` is *substantially* larger than the handed-out version, you probably included something that you shouldn't have.

For the warm-up part, just put your function definitions in `code/part1/src/Warmup`*X*`.hs`, where indicated.

For the main part, your code must be placed in the file `code/part2/src/BoaParser.hs`. It should only export the requested functionality. Any tests or examples should be put in a separate module under `code/part2/tests/`. For inspiration, we have provided a very minimalistic (and far from complete) test suite in `code/part2/tests/Test.hs`. If you are using Stack (and why wouldn't you be?), you can build and run the suite by `stack test` from the directory `code/part2/`.

The definitions for this assignment (e.g., type `Exp`) are available in file `.../src/BoaAST.hs`. You should only import this module, and not directly copy its contents into `BoaParser`. And of course, *do not* modify anything in `BoaAST`.

Content As always, your code should be appropriately commented. In particular, try to give brief informal specifications for any auxiliary "helper" functions you define, whether locally or globally. On the other hand, avoid trivial comments that just rephrase in English what the code is already clearly saying in Haskell. Try to use a consistent indentation style, and avoid lines of over 80 characters, since those will typically be wrapped in printed listings (or if someone uses a narrower editor window than you), making them hard to read.

You may (but shouldn't need to, for this assignment) import additional functionality from the core GHC libraries only: your solution code should compile with a `stack build` issued from the directory `code/part`*n*`/`, using the provided `package.yaml`. For your *testing*, you may use additional relevant packages from the course-mandated version of the Stack LTS distribution. We strongly recommend using `Tasty` for organizing your tests.

In your test suite, remember to also include any relevant *negative* test cases, i.e., tests verifying that your code correctly detects and reports error conditions. Also, if some functionality is known to be missing or wrong in your code, the corresponding test cases should still compare against the *correct* expected output for the given input (i.e., the test should *fail*), not against whatever incorrect result your code currently returns.

For this assignment, we have *not* made any provisions for white-box testing; you should test your parser through the exported `parseString` interface only. If you believe that this prevents you from adequately testing some parts or aspects of your code, feel free to remark on any *specific* concerns in the assessment section of your report. Note that we *may* try running your test suite on other implementations of the `BoaParser` module, both correct and incorrect.

Your code should ideally give no warnings when compiled with `ghc(i) -W`; otherwise, add a comment explaining why any such warning is harmless or irrelevant in each particular instance. If some problem in your code prevents the whole file from compiling at all, be sure to comment out the offending part before submitting, or all the automated tests will fail.

3.2 Report

In addition to the code, you must submit a short (normally 1–2 pages) report, covering the following two points, for the main (not warm-up) part only:

- Document any (non-trivial) *design* and *implementation* choices you made. This includes, but is not necessarily limited to, answering any questions explicitly asked in the assignment text (marked with `***` in the margin, for extra emphasis). Focus on high-level aspects and ideas, and explain *why* you did something non-obvious, not only *what* you did. It is rarely appropriate to do a detailed function-by-function code walk-through in the report; technical remarks about how the functions work belong in the code as comments.

- Give an honest, justified *assessment* of the quality of your submitted code, and the degree to which it fulfills the requirements of the assignment (to the best of your understanding and knowledge). Be sure to clearly explain any known or suspected deficiencies.

  It is very important that you document on what your assessment is based (e.g., wishful thinking, scattered examples, systematic tests, correctness proofs?). Include any automated tests you wrote with your source submission, make it clear how to run them, and *summarize* the results in the report. If there were some aspects or properties of your code that you couldn't easily test in an automated way, explain why.

  We strongly suggest (and may later mandate) that you structure your assessment into the following subsections/paragraphs:

  Completeness Is all the asked-for (as well as optional) functionality implemented, at least in principle, even if not necessarily fully working? If not, do you have any *concrete* ideas for how to implement the missing parts?

  Correctness Does all implemented functionality work correctly, or are there known bugs or other limitations? In the latter case, do you have any ideas on how to potentially address those problems?

  Efficiency Does the runtime *time* and *space* usage of your code (as you would expect it to be executed by Haskell; you don't need to actually benchmark it) reasonably match, at least asymptotically, what one would naturally assume from a proper implementation? If not, do you have ideas on how to non-trivially improve the performance of your code?

  Robustness Where relevant (which might be nowhere), how does your code behave when used *out of spec*, i.e., when given inputs that may be Haskell-type-correct, but are still illegal/invalid for some more complicated reason? *By definition*, there is no prescribed "correct" behavior for such cases, but it should preferably still be "reasonable": stopping with an informative `error` may be sensible; crashing out with a Haskell pattern-match error or similar is probably not. (Note that, for lexical/syntactic errors in the *input Boa program* (say, a malformed string constant), the correct behavior of the parser *is* specified, namely to return a suitable `ParseError`.)

  Maintainability Are common code snippets reasonably shared through parameterized auxiliary definitions, or is there a lot of code duplication in the form of copy-pasted segments with minor changes? (Note that this concern potentially also applies to your test suite!) Is the code otherwise in what you would consider in good shape (properly laid out, commented, etc.)?

  Other Anything else you consider worth mentioning, both positive and negative.

  The first two points should be substantiated by reference to the results of your formal testing. For the others, you should also justify your assessment by relevant examples or other evidence.

Your report submission should be a single PDF file named `report.pdf`, uploaded along with (not inside!) `code.zip`. The report should include a listing of your code and tests (but not the already provided auxiliary files) as an appendix.

3.3 Timesheet

In this year's run of AP, we have particular focus on student workload in the course. To help us get a more detailed, complete, and timely picture than what's provided by the formal course evaluation at the end of the block, we ask you to fill in a short timesheet for each week, detailing how much time you spent on the various parts and aspects of the assignment, as well as on other course-related activities.

The timesheet template is located in a separate file, `timesheet.txt`, in the main `code` directory, and is meant to be machine-processed, so it's important that you fill it out properly. In particular, for each time category, you should report the time you spent in hours and/or minutes, in a natural format, e.g., "`2 h`", "`15m`", "`120 m`", "`1h30m`", etc. It is particularly important that you remember to include the units, as there is no default. And try to be as precise as you can for activities on which you spent less than an hour or so; don't just round them up to the nearest hour.

If you don't know or remember how much time you spent in a particular category, give your best estimate. If you have no meaningful number for one or more categories, or prefer not to say, just write the time as a single "`x`". If you are submitting the assignment as a 2-person group, report the *average* time usage in each category (i.e., the sum of your individual contributions, divided by 2). Finally, if you have previously followed AP (or know the material from elsewhere), it is far more useful for us to know the *actual* time you spent on the course this week, than your estimate of how much time you *would* have needed if seeing everything for the first time.

You may include comments (starting with "`#`" and running till the end of the line) explaining or elaborating on your numbers, especially if you think they may be atypical. However, the timesheets are mainly intended for automated processing, so any textual comments may not get systematically registered. If you have any important points or observations that you want to bring to our attention, you should address them directly to a member of the teaching staff.

To help ensure a consistent interpretation of the categories, please use the following guidelines for accounting for the various course-related activities:

Assignment (Not an actual category. Leave as the number `3` for identification).

Installation Time spent on getting GHC/Stack installed and running on your platform. Do not include non-essential setup tweaks (such as getting syntax highlighting to work in your favorite editor, let alone trying to get any fancy IDE features configured properly; those will *not* be needed for this course.) This category will probably be zero for Assignment 3, assuming that you did the necessary installation and setup for Assignment 1 and/or 2.

Reading Time spent on the recommended readings for this week, even if not directly relevant for the assignment, as well as any supplementary reading (possibly from other than the suggested sources) that you did specifically to make progress on the assignment.

Lectures Time spent on attending the lectures. This could be less than the nominal 4 hours/week, if you skipped (parts of) a lecture for whatever reason.

Exploration Time spent on Haskell programming in *non-mandatory* activities, whether based on the suggested exercises or your own experiments. This also includes work on any *optional* parts of the assignment, regardless of whether you ultimately hand it in for feedback.

Warmup Time spent on the warm-up part of the assignment.

Development Time spent on writing and debugging the code in the main part of the assignment, *including* any integrated or ad-hoc testing you did during development.

Tests Time, *above and beyond* the above development time, dedicated specifically to *documenting* your testing in the form of an automated test suite. Note that this could be (close to) zero, if you already wrote the relevant test cases before, or together with, the code.

Report Time for writing up your design/implementation decisions and assessment.

Other Any other course-related activities not properly covered by the above categories. If you report significant time here, it would helpful to include a brief explanatory note in a comment.

If anything in the above directions is unclear (and would significantly affect the numbers you report), ask for clarification on the forum. Note that the categories and/or their descriptions may be adjusted for the later assignments, as needed.

Your timesheet numbers (or explicit lack thereof) will not affect your assignment grade, or be commented on by the TAs. But we hope that you will answer as accurately and completely as you can, to help us get a proper sense of what activities contribute significantly to the AP workload, and where specific corrective actions may be indicated for later assignments and/or future runs of the course.

You *do not* have to time yourself with a stopwatch or similar, but do try to account for any significant pauses or interruptions (e.g., lunch breaks) in blocks of time nominally dedicated to particular activities. You *may* find a dedicated time-tracking tool (such as `clockify.me`, or any number of free apps) useful and informative in general, not only for AP.

3.4 General

Detailed upload instructions, in particular regarding the logistics of group submissions, can be found on the Absalon submission page.

We also *expect* to provide an automated system to give you preliminary feedback on your planned code submission, including matters of form, correctness, style, etc. You are **strongly advised** to take advantage of this opportunity to validate your submission, and – if necessary – fix or otherwise address (e.g., by documenting as known flaws) any legitimate problems it uncovers.

Note, however, that passing our automated tests is *not* a substitute for doing *and documenting* your own testing. Your assessment must be able to stand alone, without leaning on the output from our tool.