Assignment 2 (main part): A Boa Parser

Version 1.0

Due: Wednesday, September 23 at 20:00

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 1, 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)
```

```haskell
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.

Augmenting 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 string constant

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

should be parsed as the following 9-character string (including a newline character after the first line, but *not* after the second):

```
fo\ob
a'r
```

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.)

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

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

```
not#comment
cool
```

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`'.
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

- 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 input string "`1 == True`" should parse as `Oper Eq (Const (IntVal 1)) (Const TrueVal)`.

  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 package provides by default.

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, as well as any other instances where you had to do something non-obvious. In particular:

- 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 1, you can now easily parse a Boa program (`-p` option), interpret an already parsed one (`-i`, like in Assignment 1), 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 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.

3 What to hand in

[The generic instructions are unchanged from the previous assignment.]

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 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/WarmupX.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 adequate) 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 saying in Haskell. Try to use a consistent indentation style, and avoid lines of over 80 characters.

If needed, you may import supplementary functionality from (only) the core GHC libraries: your solution code should compile with a `stack build` issued from the directory `code/partn/`, 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 recommend `Tasty`.)

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.

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 2–3 pages) report, covering the following two points, for the main (not warm-up) part only:

- Document any (relevant) *design* and *implementation* choices you made. This includes, but is not limited to, answering any questions explicitly asked in the assignment text. 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), as well as – where relevant – general issues of efficiency, robustness, maintainability, etc. 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.

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 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, style, correctness, 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 the 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.