FIT2102 Programming Paradigms 2023

Assignment 2: Parser and Transpiler

Due Date: 20/10/2023
Weighting: 30% of your final mark for the unit
Interview: SWOTVAC + Week 13
Overview: Students will work **independently** to create a parser for a subset of the **JavaScript Language** using functional programming techniques. Programs will be implemented in Haskell. **The goal is to demonstrate a good understanding of functional programming techniques as explored throughout the unit**, including written documentation of the design decisions and features.

Submission Instructions

Submit a zipped file named <studentNo>_<name>.zip which extracts to a folder named <studentNo>_<name>

- **It must contain all the code that will be marked** including the **report** and ***all code files***
- It should include sufficient **documentation** so that we can appreciate everything you have done (readme.txt or other supplementary documentation)
- You also need to include a report describing your design decisions. The report must be named **<studentNo>_<name>.pdf**.
- Only Haskell built in libraries should be used (i.e. no additional installation should be required)
    - You may use additional libraries iff for testing purposes.
- **Before zipping, run `stack clean --full` (to ensure a small bundle)**
- **Make sure the code you submit executes properly.**

The marking process will look something like this:

1. Extract **<studentNo>_<name>.zip**
2. Copy the **submission** folder contents into the assignment code bundle submission folder
3. Execute `stack build`, `stack test`, `stack run`

Please ensure that you test this process before submitting. Any issues during this process will make your marker unhappy, and may result in a deduction in marks.

Late submissions will be penalised at 10% per calendar day, rounded up. Late submissions more than seven days will receive zero marks and no feedback.

Table of Contents

Introduction

In this assignment, we will use Haskell to develop a transpiler that parses strings representing a subset of the JavaScript language. This can be converted into an Abstract Syntax Trees (ASTs) to allow for functionality like pretty printing.

This assignment will be split into multiple parts, of increasing difficulty. You are welcome to use any of the material covered in the previous weeks, including solutions for tutorial questions, to assist in the development of your parser. **You must reference or cite ideas and code constructs obtained from external sources**, as well as anything else you might find in your independent research, for this assignment.

The language you will parse will be based on the JavaScript language, however with additional restrictions to reduce ambiguity. It is important that you read the requirements of each exercise carefully, to avoid unnecessary work.

Goals / Learning Outcomes

The purpose of this assignment is to highlight and apply the skills you have learned to a practical exercise (parsing):

- Use functional programming and parsing effectively
- Understand and be able to use key functional programming principles (HOF, pure functions, immutable data structures, abstractions)
- Apply Haskell and FP techniques to parse non-trivial Javascript programs

Scope of assignment

It is important to note that **you are not writing a JavaScript interpreter**. Rather, you are only required to **parse** an expression into the necessary data types, and pretty print the resulting data type, such that it can be executed by an existing interpreter. You will **not** be required to execute the JavaScript code given, nor calculate the result (if any) of running the given code.

Exercises (26 marks)

These exercises provide a structured approach for creating the beginnings of a transpiler.

- **Part A:** parsing JavaScript expressions
- **Part B:** parsing JavaScript statements
- **Part C:** extending the interpreter to handle tail call optimisation for recursive functions
- **(Extension) ~~Part D~~ Part E:** extensions for bonus marks!

The marks for each exercise is split between parsing into an intermediary representation, and pretty printing.

You must parse the input into an intermediary representation (ADT) such as an Abstract Syntax Tree to receive marks.

Example Scripts

For each of these exercises, there will be a series of provided Javascript files. By running `stack test` it will try to parse the scripts, and save the output to a folder. `npm run dev` can be used to check the output of your pretty printer for valid Javascript code. This will test your code on the examples located within the `javascript/inputs` folder and produce output to `javascript/output` folder. You can manually inspect the output to see if it matches the formatting and by navigating to the webpage, we will be doing basic tests to make sure the prettified code is valid.

During marking, we will be running your parser and pretty printing on more complex examples than the provided example scripts, therefore, it is important you devise your own test cases to ensure your parser is valid on more complex javascript.

Furthermore, we will **not** be testing that the output of your code matches the specific formatting, and only testing if the outputted Javascript code is syntactically valid and in some cases produces correct code.

Key definitions

Multiline and Inline

Multiline means there is at least one “\n” character in the prettified output (note that “\n” is **not** a valid character inside strings, so there are no multiline strings).

Inline means there is **no** “\n” character in the prettified output.

Statements, expressions, or structures that are indicated as “can be multiline” should be pretty-printed in its multiline form if it meets the **multiline condition**.

By default, the **multiline condition** is as follows:

- It will be pretty-printed as multiline if:
    - The prettified output contains **strictly more than 42 characters**, or
    - There is **more than one immediate child** structure, statement, or expression, or
    - A **child** or **descendant** structure, statement, or expressions is **multiline**
- Otherwise, it will be pretty-printed as inline

Part A: (8 marks)

By the end of this section, you will be able to process simple JavaScript expressions.

Exercise 1 (3 marks): Parsing Integers, Strings and Boolean literals

Create the following parsers:

- A parser for integers (e.g. 4, -1, 0, 10, 403, -1203)
    - This should work for both positive and negative integers. You do **not** need to handle any floating point numbers.
- A parser for strings containing ASCII characters (e.g. “hello world”, “FIT2102 is so fun!”, “I love Haskell <3”)
    - All strings will be contained in **double quotations** (“) and you do **not** have to support the single quotation (‘).
    - You can assume there are no special characters or escape characters in the string.
- A parser for boolean literals (e.g. true, false)
- A parser for lists of these data types (e.g. [1,2,3], [true, false, 1]).
    - You do not need to check the list items are of the same type
    - You do not need to handle nested lists, and can assume lists only contain one of integers, booleans, strings.

Pretty printing these expressions means to remove any whitespace surrounding the expression, except for lists, where there is a space after each comma.

- There will only be one expression
- Examples
    - 123
        - 123
    -   123
        - 123
    - "abc"
        - "abc"
    - true   123
        - X invalid input because there are two expressions
    -    [1, 2 ,3]
        - [1, 2, 3]

Exercise 2 (3 marks): Parsing Unary/Binary Operator Expressions

All expressions involving an operator will be enclosed in brackets, including the outermost expression. This means **precedence** and **associativity** can be ignored, as they will be explicitly defined using brackets.

Create the following parsers:

- A parser for logical expressions using simple logical operators
    - &&, ||, !

- Examples
    - (!true)
    - ((!false) && true)
    - ((!false) || true)
- A parser for arithmetic expressions using simple arithmetic operators
  - +, -, *, **, /
  - Examples
    - (4 + 2)
    - (1 - (2 + 5))
    - (4 * 1)
    - ((5 + 6) / 5)
- A parser for comparison expressions using simple comparison operators.
  - ===, !==, >, <
  - Examples
    - (1 === 2)
    - (1 > 2)
    - (true === 1)
    - ("abc" > 5)
    - (((4 + 3) * 2) === (4 + (3 * 2)))

All of these operators can operate over any type.

Note that you do not need to evaluate these expressions for full marks.

Implement the functions that pretty print logic, arithmetic, and comparison operators:

- Pretty printing these expressions means:
  - There must be no more than 1 space between any expressions or literals.
  - There must be a space on either side of the +, -, &&, ||, ===, !=, >, < operators
  - There must be no more than 1 space on either side of any round brackets
  - There must be **no** space after the ! operator
- Examples
  - `(!   true)`
    - `(!true)`
  - `((!false)&&true)`
    - `((!false) && true)`
  - `(((4 +3)* 2) ===(4 +(3    * 2)))`
    - `(((4 + 3) * 2) === (4 + (3 * 2)))`

Exercise 3 (2 marks): Parsing Ternary Expressions

All expressions involving the ternary operator will be enclosed in brackets, including the outermost expression. This means **precedence** and **associativity** can be ignored, as they will be explicitly defined using brackets.

Create the following parser:
- A parser for ternary expressions
  - (<expression> ? <expression> : <expression>)
  - Examples

        - `(1 ? 2 : 3)`
        - `(true ? 1 : 2)`
        - `((1 > 2) ? (1 + 2) : (3 + 4))`

The parser should be able to handle nested ternary statements, for example:

- ((1 > 2) ? ((2 > 3) ? 0 : 1) : 1)

Implement the functions that pretty print ternary expressions:

- Pretty printing these expressions means:
    - If it meets the **multi-line condition**,
        - there must be a **newline character before**, and **space** after, the ? and : characters
    - Otherwise,
        - there must be a **space before and after** the ? and : characters
    - **Note** that ternary statements have **no immediate children** as there cannot be code blocks. The three expressions in the ternary are **not** counted towards its children.
- Examples
    - `(1?2:3)`
        - `(1 ? 2 : 3)`
    - `((1 >   2)     ?(1     + 2)    : (3+ 4))`
        - `((1 > 2) ? (1 + 2) : (3 + 4))`
    - `((((4 +3)* 2) ===(4 +(3    * 2))) ? ( ( ! false)&&true) :0)`
        - `((((4 + 3) * 2) === (4 + (3 * 2)))\n? ((!false) && true)\n: 0)`
        -
          ```
          ((((4 + 3) * 2) === (4 + (3 * 2)))
          ? ((!false) && true)
          : 0)
          ```

Part B: (8 marks)

By the end of this section, you will be able to process simple JavaScript statements and structures.

Exercise 1 (2 marks): Parsing const declarations

The variable names for const declarations will consist only of ASCII characters [a-Z0-9_]. You do not have to enforce variable naming requirements.

You should not parse mutable declarations, such as `let` or `var,` this parser will only support constants, therefore, improving the original Javascript.

There may be many such statements in a row. **Statements will always be terminated with a semicolon**.

Create the following parser:

- A parser for variable initialisation with `const`
  - Note that the RHS of the `=` can be any expression defined in Part A. *where the expression may include another variable name.*
  - Examples
    - `const aVariable = 4;`
    - `const a2_3aBcD = 1; const b = 2;`
    - `const aTernary = (true ? 1 : 2);`

Implement the functions that pretty print const variable declarations:

- Pretty printing these statements means:
  - There must be a single space after the `const`
  - There must be a single space before and after the =
  - There must be no space before the ;
  - Each statement must be on its own line
- Examples
  - `const      aVariable=4         ;`
    - `const aVariable = 4;`
  - `const a2_3aBcD=1;    const     b =2    ;`

    -
      ```
      const a2_3aBcD = 1;
      const b = 2;
      ```

Exercise 2 (4 marks): Parsing blocks

A code block is anything between two curly brackets, and consists of some number of statements (possibly zero). The code block can exist by itself or be attached to other structures (see later exercises).

Create the following parser:

- A parser for code blocks:
  - Examples
    - `{    }`

        - `{ const variable = 1; }`
        - `{ const variable = 1; const variable2 = 2; }`

Implement the functions that pretty print code blocks:

- Pretty printing these statements means:
    - Each statement or structure in the code block must be on its own line
    - If it meets the [multi-line condition](#),
        - There must be a newline **after the first** open curly bracket, and **before** the first closed curly bracket (i.e. the block must be multi-line)
        - The contents of the code block all be indented one level
    - Otherwise,
        - there must be a space inside the curly brackets (i.e. inline)
- Examples
    - `{    }`
        - `{ }`
    - `{    const variable=1    ;       }`
        - `{ const variable = 1; }`
    - `{ const a = 1; if (a) { const b = (a + 1); } }`

        -
          ```
          {
            const a = 1;
            if (a) { const b = (a + 1); }
          }
          ```
    - `{ const variable = 1; const variable2 = 2; }`

        -
          ```
          {

            const variable = 1;

            const variable2 = 2;

          }
          ```

Exercise 3 (2 marks): Parsing conditional structures

Conditional structures will **always have curly brackets** following the expression, but the else statement is **optional**.

- `if (true) { } else {} // Valid`
- `if (true) // Invalid`
- `if (true) { } // Valid`
- `if (true) { } else // Invalid`

Note that inside the curly brackets may be multiple statements or additional if/if-else structures.

Note that because of [Exercise 2 (3 marks): Parsing Unary/Binary Operator Expressions](#), any expression with a binary or ternary operator will have two sets of outer brackets.

Create the following parser:

- A parser for simple if/if-else structures:
    - Examples
        - `if (true) {}`
        - `if (true) {} else { const a = 1; }`

        -
          ```
          if ( (true && false) ){
            const a = 1;
            const b = (a + 1);
          }
          ```

Implement the functions that pretty print if/if-else structures:

- Pretty printing these statements means:
    - There must be a space **before, after, and inside** the round brackets
    - There must be a space after the `if`
    - There must be a space before and after the `else`
    - If it meets the **multi-line condition** or any of the code blocks meets the multi-line condition:
        - All the code blocks must be multi-line
        - There must be an **extra newline** before the **first closing curly bracket**
    - Otherwise,
        - All the code blocks must be inline
    - **Note** that the **immediate children** of the if/else statement is the **sum** of the immediate children in its code blocks
- Examples
    - `if (true) {       }`
        - `if ( true ) { }`
    - `if (1) { const a = 1; } else { const b = 2; }`

        -
          ```
          if ( 1 ) {
            const a = 1;

          } else {
            const b = 2;

          }
          ```
    - `if(true   ) {}else{const a =1     ;  }`
        - `if ( true ) { } else { const a = 1; }`
    - `if (1){const a = 1;const b = 2;   }`

        -
          ```
          if ( 1 ) {
            const a = 1;
            const b = 2;

          }
          ```

    -
      ```
      if ((true && false)){

        const a = 1;
       }
                 else {
      ```

```
  const b = 2;
  if (true) {
  const c = (b + 1); }
}
```

-
  ```
  if ( (true && false) ){ // note the spaces inside
    const a = 1;

  } else { // note the extra newline before this
    const b = 2;

    if ( true ) { const c = (b + 1); }
  }
  ```

Part C: (10 marks)

This section is dedicated to finally doing something more interesting with our intermediary representation. Currently, JavaScript engines do not do tail call optimisation, so we will be implementing it ourselves!

Remember that we only need to *parse* and *pretty-print,* but **not** execute the code. In the context of this section, this means we will be processing our intermediary data structure to *restructure* it into a tail call optimised form, and pretty print that.

Exercise 1 (1 marks): Parsing function calls

A function call is any variable identifier followed by round brackets that is not part of a code structure. *These function calls should be able to be assigned to const declarations (Part B.I), or contain expressions (Part A)*

Create the following parser:

- A parser for function calls:
    - Examples
        - `a();`
        - `A       ("5");`

        -
          ```
          A
          (1, 2, 3)
          ;
          ```
        - `const b = A(1,2,3)`

Implement the functions that pretty print function calls:

- Pretty printing these statements means:
    - There must be no space between the function name and the round brackets
    - There must be no space between the round brackets and the semicolon
    - There must be a space after each comma in the parameter list
- Examples
    - `a ()    ;`
        - `a();`
    - `A      (("5"    + "five"));`
        - `A(("5" + "five"));`

    -
      ```
      A
      ( (1 + 2) ,    2,
         3)
      ;
      ```
        - `A((1 + 2), 2, 3);`

Exercise 2 (3 marks): Parsing function structures with return statement

A function structure consists of the “function” keyword followed by a name (see Exercise 1 (3 marks): Parsing const declarations), any number of parameters in round brackets, and a code block.

Inside the code block for a function structure, there may be any number of “return” statements (including zero), consisting of the “return” keyword, followed by an expression, terminated with a semicolon.

Anonymous functions, assigning functions to variables, and arrow functions are **outside the scope** of this exercise.

Note that function calls are a valid expression for this exercise, including recursive calls (again, remember you should **not** have to execute the code or otherwise reason about its validity).

Create the following parser:

- A parser for function structures:
  - Examples
    - `function a(x, y, z) {}`
    - `function a() { const b = 2; }`

    -
      ```
      function somestring() {
        const a = 1;
        return (a + other());
      }
      ```

Implement the functions that pretty print function structures:

- Pretty printing these statements means:
  - There must be a space before, and after the round brackets
  - There must be a space after the function keyword
  - If it meets the **multi-line condition**,
    - The code block must be multi-line
  - Otherwise,
    - The code block must be inline
- Examples
  - `function a(){}`
    - `function a() { }`
  - `function       a(){const b = 2;}`
    - `function a() { const b = 2; }`

  -
    ```
    function somestring() {const a = 1;


      return (a + other());
    }
    ```

    -
      ```
      function somestring() {
        const a = 1;
      ```

```
                                return (a + other());
                              }
```

Exercise 3 (4 marks): Check that a function is tail recursive

Parse a function and check that is tail recursive.

The parser created for this exercise **should also** be able to parse everything in the previous exercises. That means, this parser will be the same as the parser in Exercise 1 (3 marks): Parsing function structures with return statement, except it will check the function structure to determine whether it is tail recursive or not.

For the purposes of this exercise, a **tail recursive function** has exactly following structure:

1. function keyword, followed by zero or more parameters.
2. One or more return statements that **do not involve the function name, nor other function calls** (i.e. only expressions involving const variables or literals)
    - These return statements can be nested inside other code blocks, for example an if/else statement.
3. **Exactly one return statement involving the function name**
    - The return statement must be at the **end** of the code block
    - The return statement consists **only of a function call expression to the enclosing function (i.e. recursive)** (and can have nested expressions in the parameter list)
        - **The expressions must not involve any function calls** (i.e. only allowed expressions involving const variables or literals)
        - The number of arguments to the recursive call must match the number of function parameters, e.g., no default parameters.

Examples:

Fibonacci
```
function factorial(n, acc) {
  if (((n < 0) || (n === 0))) { return acc; }
  return factorial((n - 1), (acc * n));
}
```

Factorial
```
function fibonacci(n, pprev, prev) {
  if (((n < 0) || (n === 0))) { return pprev; }
  if ((n === 1)) { return prev; }
  return fibonacci((n - 1), prev, (pprev + prev));
}
```

This example **is not** tail recursive. It fails criteria 3, as the return statement consists of two tail recursive calls.

Non Tail Recursive Fibonacci

```
function fibonacci(n) {
  if (((n < 0) || (n === 0))) { return 0; }
  if ((n === 1)) { return 1; }
  return (fibonacci((n - 1)) + fibonacci((n - 2)));
}
```

Exercise 4 (2 marks): Performing tail call optimisation (refactoring)

When pretty-printing a function:

- **If it is tail recursive, the body will be modified into a while loop, and**
- **If it is not tail recursive, it will be printed as is.**

The while loop function that is created should have the following form:

- Begin with while (true) followed by a code block, where the code block is the **same** as the tail recursive function code block except:
  - The final return statement is replaced with **deconstructing variable assignment of the parameters.** Refer to the examples below for what this looks like.

Examples:

Factorial

```
function factorial(n, acc) {
  if (((n < 0) || (n === 0))) { return acc; }
  return factorial((n - 1), (acc * n));
}
```

```
function factorial(n, acc) {
  while (true) {
    if ( ((n < 0) || (n === 0)) ) {
      return acc;
    }
    [n, acc] = [(n - 1), (acc * n)];
  }
}
```

Fibonacci

```
function fibonacci(n, pprev, prev) {
  if (((n < 0) || (n === 0))) {return pprev;}
  if ((n === 1)) { return prev; }
  return fibonacci((n - 1), prev, (pprev + prev));
}
```

```
function fibonacci(n, pprev, prev) {
  while (true) {
    if ( ((n < 0) || (n === 0)) ) {
      return pprev;
    }
    if ((n === 1)) { return prev; }
    [n, pprev, prev] = [(n - 1), prev, (pprev + prev)];
  }
}
```

Part E (up to 6 bonus marks): Extension

Implement anything that is interesting, impressive, or otherwise “shows off” your understanding of Haskell, Functional Programming, and/or Parsing.

To achieve the maximum amount of bonus marks, the feature should be similar in complexity to Exercise 3 (6 marks): Performing tail call optimisation (refactoring).

The bonus marks only apply to **this assignment,** and the final mark for this assignment is **capped** at 30 marks (100%). This means you cannot score more than 30 marks or 100%.

Some suggestions for extensions of varying complexity and difficulty:

- Convert if/then/else to ternary ? :
- Convert for loop over array to forEach/map
- Compile time optimisations (evaluating expressions)
    - Evaluating arithmetic and boolean expressions
    - Evaluating functions on inputs
        - Pure and impure
    - Fully evaluating the entire program
        - E.g. pre-computed lookup tables
- Macros (replacing certain symbols with code or expressions)
    - Compile time replacement of code, metaprogramming
    - Constant values and expressions
    - Macro functions
    - Macro code
- Convert between imperative and declarative (auto-decomposing functions)
- **Comprehensive** test cases over the parser and pretty printing
    - Warning: It is super hard to be comprehensive, stay away unless you love testing.

(Choosing one of the simpler suggestions to implement may not receive the maximum available marks).

Report (2 marks)

You are required to provide a report in PDF format of max. 600 words (markers will not mark beyond this word limit), description of extensions can use up to 200 words per extension feature.

Make sure to summarise the intention of the code, and highlight the interesting parts and difficulties you encountered. Focus on the **"why" not** the **"how".**

Additionally, just posting screenshots of code is **heavily discouraged**, unless it contains something of particular importance. Remember, markers will be looking at your code alongside your report, so we do not need to see your code twice.

Importantly, this report must include a description about why and how parser combinators helped you complete the parsing. This may involve referring to the BNF provided, or descriptions of any changes made to the provided BNF

In summary, your report should include the following sections:

- Design of the code (including data-structures)
  - High level description of approach
  - High level structure of code
  - Code architecture choices
- Parsing
  - BNF grammar
  - Usage of parser combinators
  - Choices made in creating parsers and parser combinators
  - How parsers and parser combinators were constructed using the Functor, Applicative, and Monad typeclasses
- Functional Programming (focusing on the **why**)
  - Small modular functions
  - Composing small functions together
  - Declarative style (including point free style)
- Haskell Language Features Used (focusing on the **why**)
  - Typeclasses and Custom Types
  - Higher order functions, fmap, apply, bind
  - Function composition
- Description of Extensions (if applicable)
  - What you intended to implement
  - What you did implement
  - What is cool/interesting/complex about it
  - This may include using Haskell features that are not covered in course content

There is some overlap between the sections, you should **avoid** repeating descriptions or ideas in the report.

Code Quality (2 marks)

Code quality will relate more to how understandable your code is. You must have readable and **functional** code, commented when necessary. Readable code means that you keep your lines at a reasonable length (< 80 characters), that you provide comments above non-trivial functions, and that you comment sections of your code whose function may not be clear.

Your functions should all be small and modular, building up in complexity, and taking advantage of built-in functions or self defined utility functions when possible. It should be easy to read and understand what each piece of your code is doing, and why it is useful. Do **not** reimplement library functions, such as map, and use the appropriate library function when possible.

Marking breakdown

The main marking criteria for each exercise is **correctness** – does it meet the requirements, and is it done in a way that aligns with the unit content and functional programming.

Correctness

You will be provided with some sample input and tests for determining the validity of the prettified javascript. In these tests, you will be able to see your prettified output and visually determine whether it is reasonable "pretty".

Tutors may run additional tests on additional inputs to measure the robustness of your code. Marks will be awarded proportionally for producing reasonable output and for reasonable implementation. It is highly recommended that you create your own tests as you go, on top of those provided, and that you consider possible edge cases.

Correctness also relates to the correctness of your approach. That is, how well you've applied concepts covered from the unit content.

You must apply concepts from the course. The important thing here is that you need to use what we have taught you effectively. For example, defining a new type and its `Monad` instance, but then never actually needing to use it will not give you marks. Note: using bind `(>>=)` for the sake of **using the** `Monad` when it is not needed will not count as "effective usage."

Most importantly, code that does not utilise Haskell's language features, and that attempts to code in a more imperative style, will not be awarded high marks.

Changelog

11/10/2023	Fixed tail recursive function example to correctly multiline if-statement
20/09/2023	Explicitly defined ternary children
17/09/2023	Fixed equality to always refer to ===. Fixed spacing in if/else examples. Added multi-line condition to standardise when something should be multi-line.
15/09/2023	Released