Assignment #4: Basic Expressions
Due: October 18, 11:59pm

Note: Your solution must be submitted in files named as follows and with the specified “primary” function for each part. Later files should include (using use) the necessary files from previous parts. Your solution must be stand-alone; as such, you must submit files used from previous assignments.

<table>
<thead>
<tr>
<th>Part</th>
<th>File Name</th>
<th>Primary Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>parser.sml</td>
<td>parse</td>
</tr>
<tr>
<td>3</td>
<td>printAST.sml</td>
<td>printAST</td>
</tr>
<tr>
<td>4</td>
<td>interpreter.sml</td>
<td>interpret</td>
</tr>
</tbody>
</table>

Overview

Educational Goals

• Syntactic Analysis – parsing
• Abstract Syntax – generalization/abstraction of concrete syntax
• Semantic Analysis – type checking, evaluation

This project requires that you implement a subset of the jsish language. The full jsish language is based on JavaScript, but differs in significant ways to balance the amount of work required for the course.

For this project you will implement an interpreter for the (majority of the) expressions in the language. This will give you a very good start on the recursive descent parsing technique and introduce semantic analysis through both dynamic typechecking and evaluation.

---

1 We can discuss the differences later in the course.
2 This is an intentional departure from JavaScript for educational purposes and changes the language considerably (though not necessarily for the worse). Type conversion may be considered later.
The jsish Syntax

The following grammar partially describes a subset of the syntax of jsish.
More will be added to this grammar in later projects. As such, at this point, some of the rules seem overly simple (you are encouraged to implement them in this form anyway in preparation for later projects).

In the EBNF below, non-terminals are typeset in **bold** font and terminals are typeset in *typewriter* font.

Grammar

```
program  →  sourceElement *
sourceElement  →  statement
statement  →  expressionStatement
expressionStatement  →  expression ;
expression  →  assignmentExpression { , assignmentExpression } *
assignmentExpression  →  conditionalExpression
conditionalExpression  →  logicalORExpression { ? assignmentExpression : assignmentExpression } opt
logicalORExpression  →  logicalANDExpression { | logicalANDExpression }*
logicalANDExpression  →  equalityExpression { & equalityExpression } *
equalityExpression  →  relationalExpression { eqOp relationalExpression } *
relationalExpression  →  additiveExpression { relOp additiveExpression } *
additiveExpression  →  multiplicativeExpression { addOp multiplicativeExpression } *
multiplicativeExpression  →  unaryExpression { multOp unaryExpression } *
unaryExpression  →  { unaryOp } opt leftHandSideExpression
leftHandSideExpression  →  callExpression
callExpression  →  memberExpression
memberExpression  →  primaryExpression
primaryExpression  →  ( expression ) | number | true | false | string | undefined
    eqOp  →  == | !=
    relOp  →  < | > | <= | >=
    addOp  →  + | -
    multOp  →  * | / | %
    unaryOp  →  ! | typeof | ~
```

The following rules complete the syntactic definition of jsish.

- A jsish program is followed by an end-of-file indicator; extra text is not legal.
- The terminal (token) “number” represents the set of numbers defined in the lexical analysis assignment. Similarly, the terminal “string” represents the set of strings.
Part 1: The Parser

Write a parsing function called `parse` that takes the name of a file to parse. This function will prime the parse and then invoke a function corresponding to the first rule in the grammar.

Write your parser using the recursive descent technique discussed in lecture. You can do this by determining the first sets (explicitly write them down, if you prefer) and then translating the grammar into parsing functions for each rule.

If an error — e.g., missing “;” in a conditional expression — is encountered in parsing, print an appropriate error message (using “TextIO.output (TextIO.stdErr, ...)”) and stop execution via “OS.Process.exit OS.Process.failure”.

From each of your auxiliary functions (those that parse each part of the grammar), you will want to return the current token. Since you will often want to return more than one value (i.e., the tree in the next part), you can include this token in a tuple.

Test your parser to see that it recognizes syntactically legal jsish “programs” and complains about syntactically illegal “programs”.

Part 2: Abstract Syntax Tree

In the provided files you will find a definition of an abstract syntax for the subset of jsish relevant to this assignment. This is expressed as a number of datatypes where each constructor reflects a “significant” part of the grammar above.

Using this abstract syntax, modify your parser so that it constructs an abstract syntax tree for the input source during parsing. This abstract syntax tree will later be used in the semantic analysis phase.

As noted in the last part of this assignment (for evaluation), all of the binary operators are left-associative. If you construct your tree in such a manner (i.e., left-leaning), then evaluation will be simplified.

Part 3: Echo

Write a function called `printAST` that takes an abstract syntax tree (AST) representing a “program” and prints the jsish “program” represented by the tree. The output must be syntactically valid according to the given grammar and equivalent to the original input source (e.g., it can differ in whitespace and parentheses).

What is the value of this function? First, it provides an introduction to the type of processing (structural recursion) that will be required for the last part of the assignment. Second, it provides a means for you to check that the AST generated by your parser is correct.

Note: This part will only be graded if the previous parts are completed. It is not sufficient to write only this function.

Part 4: Semantic Analysis

Write a function called `interpret` that takes the name of a file containing a “program” to evaluate. This function must invoke your parser to generate an abstract syntax tree and then invoke another function to evaluate the represented “program”. You will define multiple helper functions to simplify this task. The evaluation rules are summarized in the Semantic Rules section below.

While evaluating the “program”, your interpreter must check that there are no type violations. If a type violation is found, print an appropriate error message (to standard error) and terminate the program via “OS.Process.exit OS.Process.failure”.

During execution of the “program”, for this project only, the evaluation of an expression statement must result in the printing of the expression and the resulting value of the expression\(^3\).

For instance, evaluation of the following “program” will result in the output shown below. Additional examples of this are given in the provided files.

---

\(^3\)Printing should use the expression printing helper function from the previous part.
Program:

\[
\begin{align*}
1 + 2; \\
7 + 5 \times 2;
\end{align*}
\]

Output:

\[
\begin{align*}
(1 + 2) &=> 3 \\
(7 + (5 \times 2)) &=> 17
\end{align*}
\]

Semantic Rules

For the most part, typechecking and evaluation will work as you would expect from programming in Java (even though there is no real relation between Java and JavaScript or jsish). Some of the details are not listed here since they are discussed in lecture. If you are unsure of any required behavior, then please ask.

Evaluation

- Binary operators are left-associative.
- Operator precedence is as defined by the structure of the grammar (e.g., multiplication has higher precedence than addition).
- The conditional expression will only evaluate either the “then” or the “else” expression, not both.
- The `typeof` operator is used to determine the type of its operand and results in a string. The resulting strings, as appropriate for the operand value, are “undefined”, “boolean”, “number”, and “string”.
- A comma expression evaluates to the value of the last expression in the comma-separated list.
- Boolean (`|` or `&&`) operators are short-circuiting (i.e., the second operand is not evaluated if the truth value can be determined from the first operand).

Dynamic Type Checking

- There are four types of values: undefined, boolean, number, and string.
- Equality operators allow operands of any type. If the types do not match, then the values are not equal. If the types do match, then equality is based on comparison of the values. In all cases a boolean value is returned.
- Arithmetic operators (except for addition) require number operands and evaluate to a number value.
- The addition operator allows two number operands (resulting in a number) or two string operands (resulting in a string).
- Relational operators require number operands and evaluate to a boolean value.
- Boolean (`|` or `&&`) operators require boolean operands and evaluate to a boolean value. If the first operand does not evaluate to a boolean value, then the second operand will not be evaluated and, as such, its type will be unknown.
- The `!` operator requires a boolean operand and evaluates to a boolean value.
- The `-` operator requires a number operand and evaluates to a number value.
- The `typeof` operator takes an operand of any type and evaluates to a string.
- The expression guarding a conditional expression must be of boolean type. The conditional expression evaluates to the type of the value resulting from the expression selected (the “then” expression or the “else” expression).
Notes

- Test data with “correct” output will be given. Your output can differ in minor ways (e.g., syntax error message format) from this “correct” output yet still be correct; it is up to you to verify your code’s correctness.

- Your program will be exercised by some shell scripts, which will be provided. These scripts depend on the exit status of your program. If your program detects a “serious” error, your program should use “OS.Process.exit” to terminate.

- Grading will be divided as follows.

<table>
<thead>
<tr>
<th>Part</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>50</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>30</td>
</tr>
</tbody>
</table>

- Get started **now** to avoid the last minute rush.