Compiler Construction/Fall 2014/Project Milestone 1*

Eva Rose
evarose@cs.nyu.edu

Kristoffer Rose
krisrose@cs.nyu.edu

Assigned Thursday 9/18/2014, due Monday 10/6/2014 at 8am

Welcome to the first project milestone for the fall 2014 compiler construction class at the NYU Courant Institute! Your task is to program a parser, which includes the lexical analyzer and top-down parser, using the HACS system (version 0.8.nyu as documented in the H1 through H3 handouts[1]). Your parser should parse the JST language, which we have given the syntax of below.

1 Assignment

To complete this milestone submit two files to the pr1 assignment in the NYU classes system:

pr1-student.hx a file with your parser that, when build with HACS, parses JST files.

pr1-student.pdf a document with project-related overview information, including but not limited to with an explanation of any issues you had to resolve, like decisions you made on points where the specification here is unclear, and limitations of your implementation.

(You may submit more files but if so their purpose must be explicitly explained in the **pr1-student.pdf** document.) The grader will run your submitted HACS script with commands like

```
make pr1-student.run
./pr1-student.run  --sort=Program  samples/Hello.jst
```

with a variety of sample programs. (Test runs will be on the NYU energon-machines with a fresh install of HACS.) A run is *successful* if the program is echoed back to the standard output (spacing differences are alright) and *fails* if the program outputs a Java exception.

Your submission will be graded on the following points:

- Whether your implementation succeeds on the correct sample JST programs and fails on the incorrect ones when run as explained above.
- Whether the HACS source code is well structured and easy to read.
- Whether any differences in what your parser accepts and what the specification below states as syntactically correct programs are properly explained in the submitted document, ideally with a plan for how discrepancies can be solved. Pay special attention to aspect of the syntax, described below, that your parser does not handle because you believe that they should be handled by the subsequent semantic analysis phase.

*Revision 2 (Friday 5th December, 2014, 22:41)

[1] All documentation available from the class web page, `http://cs.nyu.edu/courses/fall14/CSCI-GA.2130-001/`

```
1  class Greeter {
2    string greeting;
3    void set(string g) {
4      this.greeting = g;
5    }
6    string greet() {
7      return "Hello, " + this.greeting + "!";
8    }
9  }
10
11 function string main() {
12   var Greeter g;
13   g = new Greeter();
14   g.set("World");
15   return g.greet();
16 }
```

Figure 1: JST greeter program.

- Whether any particularly difficult parts of your **pr1-student.hx** are properly explained in your **pr1-student.pdf** document.
- Whether the documentation demonstrates that the project was planned and executed in a sensible and timely manner.
- Whether the documentation explains what tests were performed, and how it can be verified by the grader that they gave the expected result.

As always, collaboration is encouraged, but make sure to carefully document who you worked with, and on what, in your submitted **pr1-student.pdf** document.

2 JST Syntax

JST is a typed variant of a subset of ECMAScript[2] (the official standard name for the language colloquially known as "JavaScript"). Figure 1 is a small sample of a JST program with line numbers, and with keywords and strings highlighted in color (we shall use this style throughout): When executed, this program will output the string `Hello, World!` (Further example programs are available through the class web site.)

A JST program is a sequence of class declarations and function definitions, where one function must be called `main` with no arguments and returning a string value, as shown, and the result of evaluation is the string composed by a call to `main()`. This section presents the JST syntax summarily, from the bottom up, starting with the tokens.

2.1 Definition. The *lexical grammar* of JST recognizes the following tokens:

1. *Identifier* tokens start with a letter, `$`, or `_`, followed by more of the same as well as digits, except that keywords (literal tokens used by the grammar) are not permitted as identifiers.

[2] *http://www.ecmascript.org/*.

2. *Integer* tokens are sequences of digits.

3. *String* tokens are sequences of characters enclosed in either `'` (single quote) or `"` (double quote). The sequence of characters can contain any individual characters except the quote used for enclosing, `\` (backslash), and newlines, and can furthermore contain a `\` (backslash) followed immediately by one of the following escapes:

   - a newline is a *line continuation*, which is ignored;
   - one of the characters `'"\` stands for that character;
   - one of the characters `nt` stands for an included newline and tabulation character, respectively;
   - an `x` followed by two hexadecimal digits stands for the character with that byte encoding.

   Note that the comment markers are not special inside strings.

In addition all JST special characters and keywords, as they appear in the rest of this specification, are recognized as separate tokens.

Finally, *Comments* of two forms are ignored: *single line* comments with all the text from a `//` up to (but not including) the closest newline, and *multi-line* comments of everything between a `/*` and a `*/`, including newlines. Multi-line comments do not nest.

The most basic construction is the *Expression*, which we define next.

2.2 Definition. The *Expression* non-terminal of JST is either a *Literal* or an *Operation*. A *Literal* is either one of the following:

- the tokens *Integer* or *String*;
- an object literal $\{k_1 : \mathit{Literal}_1, \ldots, k_n : \mathit{Literal}_n\}$ for $n \geq 1$ with each key k_i an *Identifier* (also called "field name").

An *Operation* captures the following forms, with E denoting an *Expression*:

```
(E)   Identifier   this   new Identifier()
E(E)   E()   E.Identifier
!E   -E   +E
E*E   E/E   E%E
E+E   E-E
E<E   E>E   E<=E   E>=E
E==E   E!=E
E&&E
E||E
E=E   E+=E
E,E
```

The *Operations* are given by order of precedence by line: operations bind tighter than operations in subsequent lines. The operations are subject to the following additional constraints:

- All the operations associate left to right, except the operations in the last three lines associate right to left, so, for example, `a.b.c` is the same as `(a.b).c` but `a=b=c` is the same as `a=(b=c)`.
- The object *Literal* form is only allowed as the right side of an assignment (`=`).

JST modifies ECMAScript by requiring explicit types on variables and parameters.

2.3 Definition. The *Type* non-terminal of JST captures the notation for structural types. A *Type* has one of he following forms

```
boolean
int
string
void
Identifier
```

Expressions are composed into *Statements*.

2.4 Definition. The following kinds of statements are permitted by JST:

```
{ Statement... }
var Type Identifier ;
;
Expression ;
if ( Expression ) Statement
if ( Expression ) Statement else Statement
while ( Expression ) Statement
return Expression ;
return ;
```

with the added convention that every `else` belongs to the closests preceding `if` not already assigned an `else`. The statements above use "..." to indicate that inside braces (`{}`) one can have zero or more statements.

Statements are used inside (top level) *Declarations*.

2.5 Definition. A *Declaration* is either a *ClassDeclaration* or a *FunctionDeclaration*. A *ClassDeclaration* takes the form

```
class Identifier { Member... }
```

where each *Member* has one of the forms

```
Type Identifier ;
Type Identifier ArgumentSignature { Statement... } ;
```

(the two kinds of members are known as *fields* and *methods*). A *FunctionDeclaration* takes the form

```
function Type Identifier ArgumentSignature { Statement... }
```

In each case the *ArgumentSignature* takes the form

```
( Type Identifier, ..., Type Identifier )
```

where the "..." indicates that there may be zero or more comma-separated *Type Identifier* formal parameters.

Finally, the *start symbol*.

2.6 Definition. The start symbol of the JST language is the *Program*, which is defined as a sequence of *Declarations* with at least one declaration.

This is the end of the JST syntax specification.