Christopher Pac
N-13242624
cpp270@nyu.edu

Compiler Construction
Spring 2015
Project Milestone 1

TABLE OF CONTENTS

1 Introduction

This document will cover the documentation requirement part specified in the Project Milestone 1. It will explain issues that were encountered during the parser implementation and the solutions that were adopted. It will also cover discrepancies between the specifications and the actual implementation.

This document is divided into four sections that naturally follows the project execution. The introduction section provides information about the general approach that was taken while working on the parser, the computing environment that was used, basic conventions that were followed, and finally outside sources that were used while working on this project.

The rest of this document covers the implementation details, general issues that were encountered, and all the tests that was done on the parser.

1.1 General Approach to the Project

As a disclaimer I would like to state that I have very limited experience with JavaScript and therefore I've taken the approach of following the grammar that was provided in the project assignment as faithfully as possible. There is especially one instance with an object literal where I found the specification to be questionable (see section 2.2 below for details). One exception, to sticking as close as possible to the specifications, is that when discrepancies occurred between the sample test files and the project definitions I opted for adopting the grammar in the sample files.

1.2 Environment

- Computing environment
  - Linux energon1.cims.nyu.edu

- HACS version
  - HACS 1.0.2

1.3 Conventions

All lexical tokens that are used in grammar productions follow the convention of having all characters in upper case. All other declarations (sorts and token fragments) follow the CamelCase convention where the first letter is always capitalized.

1.4 Credits

The following outside websites were used as reference:

- http://www-archive.mozilla.org/js/language/js20-2002-04/formal/lexer-grammar.html
- http://www-archive.mozilla.org/js/language/js20-2002-04/formal/parser-grammar.html

2 Implementation

This section covers the specified project definitions. It explains how theses definitions were implemented, how encountered ambiguities were resolved, and provides code clarification when needed. This section is conveniently divided the same way that the project definitions are divided and it also reflects the source code breakdown in the Pr1ChrisPac.hx file.

Not all the section will have detail information about the implementation. This is because most of it is fairly easy to deduce from the actual source code, thus only major highlights, discrepancies, difficulties, and solutions will be discussed.

2.1 Lexical Grammar

As described in 1.3 Conventions section, tokens that are used in grammar productions are in all upper case and have also been shortened, thus ID is Identifier, INT is Integer, and STR is String. The regular expressions for those tokens are straight forward and follow the many examples provided in the HACS manual. The only thing thats worth mentioning is that HACS manual states that the characters `\ ' "` need to be always escaped.

The only somewhat complicated regular expression is the one for the multi-line comments. Its designed to allow any number of '*' inside the comments. This allows for comments like this:

```
/****** Hello
*this is a *******
*comment ******/
```

Furthermore, the comments are ignored through the use of the special HACS `space` declaration and are thus discarded.

2.2 Expression

This project definition was by far the most difficult and lengthy section to implement. It was also the only one with the most glaring ambiguity. The one thing that was of tremendous help was HACS convenient production declaration syntax. This convenience allowed for easy way to specify precedence and left-right recursion through the use of the @-annotation, which left the actual resolution to HACS to figure out.

The *Expression* production follows the '`sort Exp`' examples provided in the lecture slides and HACS manual except that it is greatly expanded to accommodate the additional productions. And just like the example, the precedence in the code decreases as we go down the production declaration, which is the reverse of the table provided in the project specification.

The *Expression* production is divided into three notable productions: *AExp*, *SExp*, *SubExpression*. The *Expression* production allows for Integer, String and Object literal to be used in it. In order to accomplish this and to disallow Object Literal to be on the left side of the assignment operation, the *Expression* production was split into *AExp* and *SExp* productions. The *SExp* (Simple Expression) production uses only the Integer and String literal and thus defines the assignment productions. The *AExp* (Advanced Expression) production is almost identical to *SExp* except that it uses Object Literal and omits the assignment productions.

Object literals are part of *Expression* production and object literals are composed of *Expressions*. This introduces a grammar ambiguity when combined with the '*E* , *E*' production (that is part of the *Expression* production). The parser is confused about the first colon in the object literal statement since it expects another expression after a comma. The error printed by HACS is that the parser expects one of the operation symbols (< > + - etc) and not a colon. Therefore, the Object literal and the '*E* , *E*' *Expressions* productions are incongruent. The solution to this problem is to reduce the scope of one of the productions in some way thus removing one of the conflicting productions. First simple solution would be to not use *Expression* in the Object literal production and only use *String*, *Integer*, or another Object literal. On some level this makes sense since some of the *Operation* productions may not be allowed by JavaScript to be in an Object literal (i.e. *E* **E*). Since I'm not sure which operations should or should not be allowed I've attempted to be as close to the specified project grammar as possible and only removed the '*E* , *E*' *Expression* production from the Object literal. This was done by defining a *SubExpression* production which is used in the Object literal and does not include '*E* , *E*' production. The final *Expression* production is thus composed of the *SubExpression* production and '*E* , *E*' production, where *SubExpression* production naturally has higher precedence. This only worked out nicely because '*E* , *E*' production has the lowest precedence, otherwise I probably would have used *Literal* (Integer, String, Object Literal) in the Object literal production.

2.3 Type

This was a simple production to implement with no ambiguities.

2.4 Statement

There are three things of note here otherwise it was a fairly easy production to implement. The first issue is whether an empty parentheses statement is allowed and is discussed in section 3 General Issues. Second, are the additional *Substatement* and *AllStatements* productions which are there only to implement recursion and to satisfy the '…' requirement where zero or more statement repetitions are allowed. This is also discussed in section 3 General Issues under regular expression repetition markers. Third, the *if* production was done by declaring an *ElseStatement* production that covers the *else* or is empty. I believe the code satisfies the requirement of pairing every *else* with its closest preceding *if* but I was not able to test this. I've made a suggestion concerning things like this in section 3 General Issues under printing a parse tree representation.

2.5 Declaration

This was a simple production to implement but I did encounter two very minor issues. The first issue is whether an empty parentheses statement is allowed and is discussed in section 3 General Issues.

The second issues was discovered during testing and it pertains to the semi-colon at the end of one of the *Member* productions.

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

One of the test files failed when the semi-colon was part of the production so I have removed it to satisfy the test.

2.6 Program

A simple recursive production with no ambiguities.

3 General Issues

This section briefly describes some general issues that were encountered while implementing/ testing the parser.

The specification does not explicitly indicate if productions that have parentheses should allow empty statements within those parentheses. For example, should empty object literal, '{ }', or empty argument signature, '( )', be allowed by the syntax grammar? I have chose to allow such empty parentheses statements since they are allowed by JavaScript and most other programming languages.

The issues that I have encountered is that I had to provide two separate productions for these empty parentheses statements. One production with a space between the parentheses and one production with no space between the parentheses: `| ⟦ ( ) ⟧ | ⟦ () ⟧ ;`

Based on the error messages printed, it appears that the production with no spaces between the parentheses is treated as a single token. I think there are two different solutions that can be implemented, unless there is some setting in HACS that will address this. One is to explicitly define a token for the opening and closing parentheses and second is to provide two productions. I have picked the second approach.

The second issue is more of a request for a feature to HACS. It would be very convenient to annotate the results of HACS productions with parentheses such that the parse tree is reflected by those parentheses. For example the statement:

if (cond1) then if (cond2) then expr1 else expr2

would be printed as (colors used for clarification):

(if (cond1) then (if (cond2) then expr1 else expr2))

Finally, it would also be convenient to use regular expression repetition markers (?, +, *) in grammar productions. For example, instead of creating two productions for the following grammar `{ Statement... }`, we could specify a single HACS production with a Kleene star. This would make the source code less cluttered.

4 Testing

This section is divided into tests that were run on the sample .jst files provided in the project and any additional custom tests that were done to check for the correctness of the parser.

All tests were done in the environment specified in section 1.2 Environment.

4.1 Sample Testing

Succeeded as required.

```
$ ./Pr1ChrisPac.run  --sort=Program samples/Comment.jst
   function  string  main  (  )  {    return     "/* Not a comment!
*/"     ;    }
```

Succeeded as required.

```
$ ./Pr1ChrisPac.run  --sort=Program samples/Fib.jst
   function  int  fib  (    int  n   )  {    if (     n  >=   0      )
return      fib (     n  -   2      )  +   fib (     n  -   1      )
;    else   return     0     ;      }      function  string  main  (
)  {    return      "Fib of 5 = "   +   fib (     5     )     ;    }
```

Succeeded as required.

```
$ ./Pr1ChrisPac.run  --sort=Program samples/FastFib.jst
```

… output too long to print

Succeeded as required.

```
$ ./Pr1ChrisPac.run  --sort=Program samples/Hello.jst
   class Greeter {    string  greeting ;     string  greet  (  )  {
return       "Hello, "   +   this .greeting   +   "!"      ;    }    }
function  string  main  (  )  {    var  Greeter  g ;          g  =
{  greeting :    "World"     }        ;      return      g .greet ()
;      }
```

Succeeded as required.

```
$ ./Pr1ChrisPac.run  --sort=Program samples/String.jst
   function  string  main  (  )  {    var  string  one ;          one
=      "111'\""        ;      var  string  two ;          two  =
'2\x082"\''        ;      var  string  three ;          three  =
""        ;      return      one  +  two   +  three     ;          }
```

Failed as required.

```
$ ./Pr1ChrisPac.run  --sort=Program samples/Comments.badjst
Exception in thread "main" java.lang.RuntimeException:
net.sf.crsx.CRSException: Encountered " <T_ID> "which "" at line 2,
column 1.
Was expecting one of:
    "class" ...
```

```
    "function" ...
```
… too long to print all the output

Failed as required.

```
$ ./Pr1ChrisPac.run  --sort=Program samples/ExtraElses.badjst
Exception in thread "main" java.lang.RuntimeException:
net.sf.crsx.CRSException: Encountered " "else" "else "" at line 4,
column 3.
Was expecting one of:
    "+" ...
```
… too long to print all the output

4.2 Custom Testing

The following additional tests should execute successfully and did so:

Object Literal

```
$ ./Pr1ChrisPac.run  --sort=Expression --term='{ fst : i>0 , snd : 1 }'
    {   fst :    i  >   0       ,  snd :    1     }
```

Assignment of an object literal

```
$ ./Pr1ChrisPac.run  --sort=Expression --term='bar = { fst : i>0 , snd : 1 ,
foo:this }'
    bar  =      {    fst :    i  >   0       ,  snd :    1      ,  foo :
this    }
```

Multiple assignments with Object literal at the end

```
$ ./Pr1ChrisPac.run  --sort=Expression --term='bar = foo = this =
{ fst : i>0 , snd : 1 , foo:this }'
    bar  =      foo  =      this  =      {    fst :    i  >
0       ,  snd :    1      ,  foo :   this    }
```

Multiple And Or operations

```
$ ./Pr1ChrisPac.run  --sort=Expression --term='True || False || True
&& False'
     True  ||  False   ||   True  &&  False
```

Parentheses inside an expression

```
$ ./Pr1ChrisPac.run  --sort=Expression --term='True || (False || True)
&& False'
    True  ||  (  False  ||  True  ) &&  False
```

return Expression Statement

```
$ ./Pr1ChrisPac.run  --sort=Statement --term='return True || False ||
True && False;'
  return      True  ||  False   ||   True  &&  False      ;
```

Multi-line comment

```
$ ./Pr1ChrisPac.run  --sort=Statement --term='return this; /*******
Hello
> * this is a ****************
> * comment ****************/'
  return    this    ;
```

The following additional tests should <u>fail</u> and did so:

Object literal on the left side of an assignment operation

```
$ ./Pr1ChrisPac.run  --sort=Expression --term='bar = { fst : i>0 , snd
: 1 , foo:this } = foo'
Exception in thread "main" java.lang.RuntimeException:
net.sf.crsx.CRSException: Encountered " "=" "= "" at line 1, column
42.
Was expecting one of:…
```

… too long to print all the output