Homework 3: ANTLR + Templating Language

Dates

Assigned: 4/21.
Due: 5/8, 11:59pm.

Overview

This assignment will implement an extensible templating language that compiles to JavaScript, using syntax similar to Handlebars (which follows the syntax of Moustache). The assignment implements the compiler iteratively, adding new capabilities and partially rewriting it on each step. In the first problem, you will write a parser for a templating language, which interleaves blocks of raw text with embedded expressions and blocks. In the second problem, you will implement a basic compiler of such templates into JavaScript functions and add an extensibility mechanism of "helpers", which allow users to add custom expressions and blocks into their templates. In the third problem, you will extend your templating language by implementing three library helpers.

Yes, you can add comments to this document! Like this. This is the best way to ask clarification questions and pose suggestions on how to improve the homework.

Before Starting

We will be testing your homework projects using Chrome 49 (not Chromium). Since different versions of Chrome behave differently, make sure you are using the right version to avoid any compatibility issues.

Supplied Skeleton

The webpage `index.html` executes all provided test cases from `test.js`. Initially, most of them will fail; your goal is to implement the compiler pipeline to make those tests pass. We will also test your code or several hidden additional tests, so please make your implementation general-purpose (as opposed to working only on the provided tests).

Chrome enforces strict permissions for reading files from the local file system (background information on wikipedia). ANTLR runtime loads autogenerated lexer and parser dynamically using require.js. Therefore, to execute your tests, you will need to run a local HTTP server. Start your own local web server by running Python's built-in server in the directory with `index.html`. (If you don't have Python, installing it is the easiest way to get a local webserver.)

```
python2 -m SimpleHTTPServer 8888
or
python3 -m http.server 8888
```

Once the server is running, direct your browser to http://localhost:8888/. The browser will display the `index.html` file with a report on test execution.

The skeleton defines two basic grammars: a *lexer grammar* in `HandlebarsLexer.g4` and a parser grammar in `HandlebarsParser.g4`. A lexer grammar describes how to split a program source into *tokens*; in this case, the tokens separate raw text and *islands* of templating constructs: expressions, blocks, identifiers, and literals. A parser grammar describes how to interpret a stream tokens as a program using *parser rules.* Currently, only the top-level parser rules are defined:

```
document : element* EOF ;

element
    : rawElement
    | expressionElement
    | blockElement
    | commentElement
    ;
```

Our Handlebars compiler is defined in `compiler.js` as an ANTLR listener. Currently, it only defines a handful of top-level event handlers — namely, a pipeline that takes as input a template string, tokenizes it using an autogenerated lexer, parses it with an autogenerated parser, and then executes a tree walk over the returned AST with itself as a listener. This tree walk will generate a JavaScript function (as a source code string) that will represent our compiled template. This function is compiled into an executable JavaScript `Function` object and returned to the caller.

`HandlebarsCompiler` is implemented as a "JavaScript 5.1 class" using prototype inheritance. Our HW2 handout includes a section on classes in JavaScript. All concepts from that handout still apply in HW3 except that you do not need to define your own classes because ANTLR has defined them for you. You only need to add methods to `HandlebarsCompiler` that override default event handler implementations in the autogenerated ANTLR listener. For example, to override a handler that executes *after* we exit an "`expressionElement`" AST node, you define the following function:

```
HandlebarsCompiler.prototype.exitExpressionElement = function (ctx) {
    // your code here
};
```

For more information, please consult the ANTLR documentation on JavaScript target. We also wrote a small ANTLR refresher below about the general concepts on ANTLR tree walking.

ANTLR refresher

An ANLTR *lexer grammar* consists of one or more *lexer rules.* A *lexer rule* defines a *token* — a single chunk of a program source string. The goal of a lexer is to split a source string into a stream of tokens. A lexer rule defines one or more *alternatives* using a limited subset of a regular expression syntax. The alternatives may include references to other tokens (called *fragment tokens*). As an example, here is a lexer rule that defines a FLOAT token with 3 alternatives, and two lexer rules that define fragment tokens INT and EXP that are referenced by FLOAT.

```
FLOAT
    :   '-'? INT '.' INT EXP?   // 1.35, 1.35E-9, 0.3, -4.5
    |   '-'? INT EXP            // 1e10 -3e4
    |   '-'? INT                // -3, 45
    ;
fragment INT :   '0' | [1-9] [0-9]* ; // no leading zeros
fragment EXP :   [Ee] [+\-]? INT ;
```

Similarly, an ANTLR *parser grammar* consists of one or more *parser rules*. A *parser rule* defines a *nonterminal* — a single type of node in a program AST. The goal of a parser is to build an AST over a stream of tokens that are produced by a lexer. A parser rules defines one or more alternatives, which may reference tokens, other nonterminals, or their combinations using a limited subset of a regular expression syntax. Alternatives may be *labeled*, in which case individual parsed alternatives are represented with their own dedicated nodes in the AST.

As an example, here is a definition of a “set of characters” in regular expressions, taken from our ANTLR tutorial.

```
set :  '[' inverse='^'? (members+=setMember)+ ']' ;

setMember returns [repr]     //  here ‘repr’ is a “rule context attribute”, explained later
    : l=CHAR '-' r=CHAR      # RangeSetMember
    | CHAR                   # CharSetMember
    ;
```

This snippet defines two parser rules (nonterminals). The `set` parser rule includes only one alternative, whereas the `setMember` parser rule defines two alternatives, labeled `RangeSetMember` and `CharSetMember`. The alternatives reference a CHAR token from the lexer grammar, as well as 4 anonymous *implicitly defined* single-character tokens `‘^’`, `‘[’`, `‘]’`, and `‘-’`.

By convention, nonterminals are named in camelCase, tokens are named in UPPERCASE, and label names are named in PascalCase.

Our `HandlebarsCompiler` is implemented as an ANTLR listener. As a reminder, the autogenerated `HandlebarsParserListener` contains empty implementations for "`enter`" and "`exit`" event handlers for every parser rule and for every labeled alternative in the grammar. For instance, for the "regex character set" example above, ANTLR will generate the following event handlers:

- `enterSet`
- `exitSet`
- `enterSetMember`
- `exitSetMember`
- `enterRangeSetMember`
- `exitRangeSetMember`
- `enterCharSetMember`
- `exitCharSetMember`

When you invoke a tree walk over an AST, ANTLR calls corresponding event handlers for any AST node it enters/exits. If you override an event handler in `HandlebarsCompiler`, your implementation will be called instead of the default one. For instance, this handler is called each time *after* the tree walk visits an `ExpressionElement` AST node.

```
HandlebarsCompiler.prototype.exitExpressionElement = function (ctx) {
    // your code here
};
```

Here `ExpressionElement` (part of the name `exitExpressionElement`) may be either a nonterminal (in which case it should be called "`expressionElement`"), or a *label* on an alternative associated with some nonterminal.

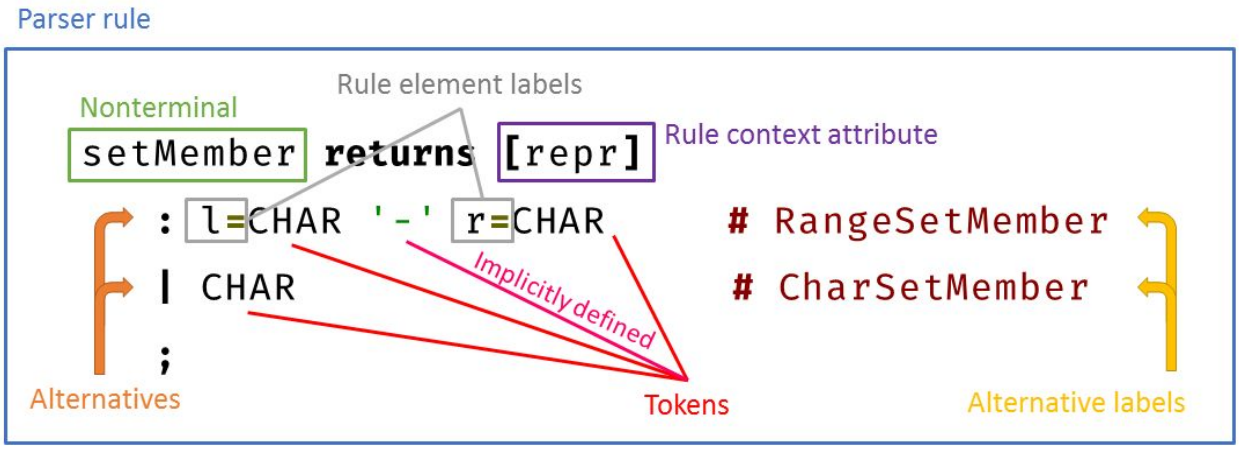
The handler receives as input a *context*, which is wrapper around an AST node. A context provides many useful APIs with information about its corresponding AST node. (The linked API description is for Java, but available methods and fields in JavaScript are exactly the same.)

To add some convenience information to a context object, you can leverage other ANTLR features: *rule element labels* and *rule context attributes*. A rule element label is attached to an element of a parser rule (a token or a reference to another parser rule). In an AST, the corresponding match of that element is captured as a field on the context object. For instance, in the "regex character set" example above, every context for a `set` AST node will contain additional fields `inverse` and `members`. The `inverse` field will be a reference to a matched `'^'` token or `null` if the caret is absent in a given AST since it is optional. The `members` field will be a list of references to matched `setMember` AST nodes. All children ASTs are also

available if you call `.getChild(i)` on the context object, but labeling them explicitly allows you to manipulate children ASTs more easily.

Rule context attributes allow you to define additional fields on the context objects. In the "regex character set" example above, "`returns [repr]`" annotation on a `setMember` nonterminal adds a `repr` field to each instance of its associated `SetMemberRuleContext` context. This feature is less relevant for a JavaScript target than it is for a Java target because in JavaScript you can always add arbitrary fields to any object at runtime. However, an explicit annotation specifying an expected field still helps readability, code completion, and documentation.

Here's a one-slide summary of all involved concepts:



For more information on ANTLR, please check out the following resources:

1. Public ANTLR documentation. It explains many of the concepts above with substantially more examples.
2. "The Definitive ANTLR 4 Reference"
   Some excerpts from the book are available online for free on the publisher's website. All three are directly relevant to this homework assignment.
   - Let's Get Meta — an excellent high-level overview.
   - Building a Translator with a Listener
   - Islands in the Stream
3. Our ANTLR tutorial, shown in class.
4. Lecture slides on ANTLR (ppt, pdf).

Interacting with ANTLR as your work on your homework

Over the course of your homework, you will need to change the provided grammars and regenerate lexer/parser/listener implementations using ANTLR (likely multiple times). Your compiler in compiler.js expects these classes to be present in the same directory as `HandlebarsLexer.js`, `HandlebarsParser.js`, and `HandlebarsParserListener.js`, respectively. Depending on your preferred environment, OS, and text editor, you may find

different methods convenient. Feel free to add more instructions for your favorite editor in this document or on Piazza!

Console

Navigate to http://www.antlr.org/ and follow the "Quick Start" instructions for your OS. They will walk you through downloading ANTLR and adding a new command `antlr4` in your environment (as a shell alias or a batch file).

Since ANTLR is written in Java, you will need a JRE installed on your machine. Your resulting Handlebars compiler does not actually need JRE to run, since it is pure JavaScript; you only need Java to execute ANTLR on your grammar files at development time.

After you install and configure ANTLR, run the following command from the `hw3/` directory:

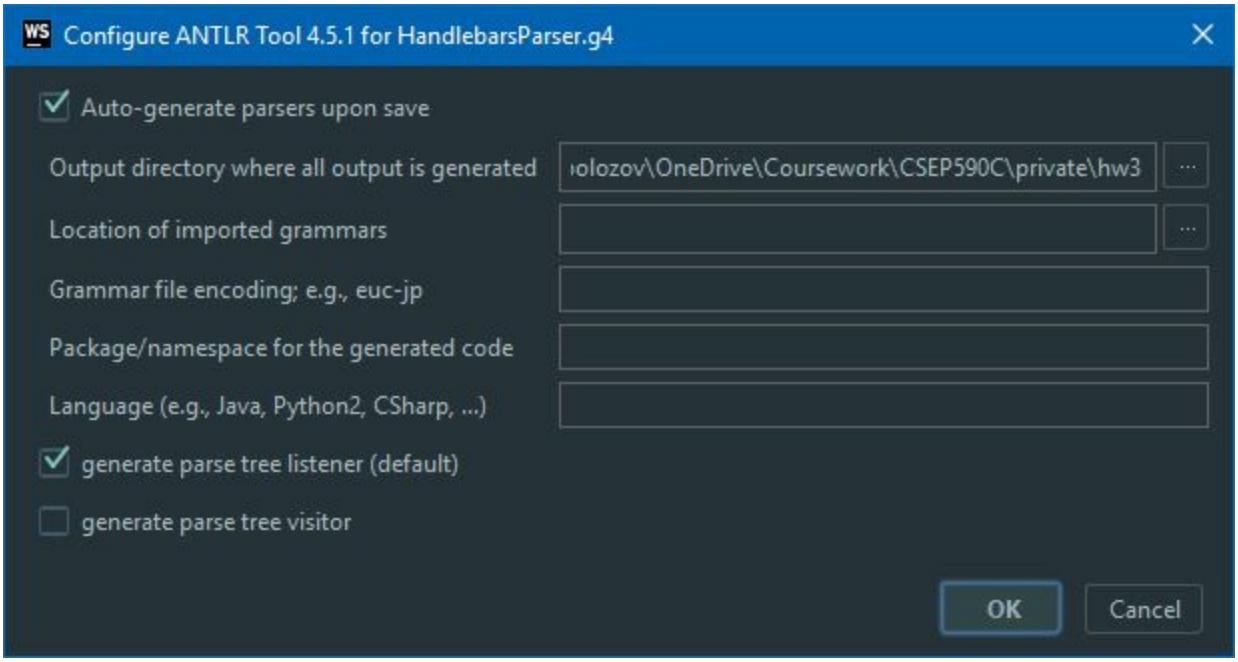
```
antlr4 -o . -listener -no-visitor -lib . HandlebarsLexer.g4 HandlebarsParser.g4
```

The required files should be automatically generated in the same directory. Repeat the command as necessary when you make any changes to the grammars.

JetBrains IDEs

ANTLR provides an extension for JetBrains IDEs (IntelliJ IDEA, WebStorm, etc.). JetBrains IDEs are cross-platform and available for free for students for educational purpose. You can use your @uw.edu account to acquire an free educational license.

After you install ANTLR extension in the IDE, open any grammar, and then configure the extension in "Tools → Configure ANTLR…" You want to change the output directory to the root directory of your project (i.e. `hw3/`) instead of the `gen/` subdirectory. You also may find the "Auto-generate parsers upon save" setting helpful.



Sublime Text

There is an extension for Sublime Text 2 and 3. It relies on a globally configured `antlr4` command and correct CLASSPATH, which is a part of ANTLR "Quick Start" instructions.

Problem 1: Grammar definition (30%)

The current parser grammar defines a general skeleton for parsing elements of a template. Each element in a template is either a piece of raw text, an *expression element*, a *block element,* or a comment. Out of these, the supplied skeleton currently defines and compiles only the raw text with comments.

> **NOTE:** The *raw text element* is currently defined simply as "`rawElement : TEXT`". This definition does not allow opening curly braces in the raw text because they are filtered out by the TEXT token. Because of this, one of the initial parsing tests, "Parse templates with complex lexing", may fail. As a warmup, try changing the definition of `rawElement` to account for possible brace characters. This should be a 2-line change in `HandlebarsParser.g4` and should work without any additional code in the compiler. You can introduce a new nonterminal.

Task A: Define expressions

First, we need to define parser rules for expressions in `HandlebarsParser.g4`. In the rest of this subsection, we formally define what constitutes a valid Handlebars expression.

> **NOTE:** here and everywhere else in the document, double quotes (“”) are not part of the syntax unless otherwise specified.

An expression element in Handlebars starts with “`{{`“, ends with “`}}`”, and contains *exactly one* of the following syntactic constructs in between.

Literals

A literal is either an integer (e.g. “`-7`”, or “`42`”), a float (e.g. “`1.35`”, “`-3e4`”, “`-4.5`”), or a single-quoted string (e.g. “`'hello\nworld'`”). We require strings to be valid JavaScript literals; in particular, standard JS escaping rules apply.

The provided lexer grammar already contains most of the building blocks for correct parsing of integer, float, and string literals: token types INTEGER, FLOAT, and STRING, respectively. **However, it’s incomplete:** one of the three token types does not cover all examples given in the previous paragraph. You will need to identify this token and make a small modification to `HandlebarsLexer.g4` to fix this.

Data lookups

A sole identifier (i.e. without any following space-separated parameters) is interpreted as a *lookup of a property in the current data context*. Identifiers are classified with a token type ID.

Parenthesized expressions

An expression surrounded with parentheses is interpreted as itself.

Helper applications

An identifier followed by non-zero space separated *parameter expressions* is interpreted as an *application of an expr helper with the given parameters.* A helper application (including all its parameter expressions) is also considered a kind of expression.

A helper application must be either a top-level expression in an expression element, or immediately surrounded by parentheses. In other words, this template:

```
{{ concat 'Hello' last_name ', ' substr first_name 0 1 '.' }}
```

should be interpreted as a call to a “`concat`” helper with the following parameters:

- A literal string “`Hello`”
- A value of a property “`last_name`” on the current data context
- A literal string “`, `”
- A value of a property “`substr`” on the current data context
- A value of a property “`first_name`” on the current data context

- A literal integer 0
- A literal integer 1
- A literal string “`.`”

In contrast, this template:

```
{{ concat 'Hello' last_name ', ' (substr first_name 0 1) '.' }}
```

should be interpreted as a call to a “`concat`” helper with the following parameters:

- A literal string “`Hello`”
- A value of a property “`last_name`” on the current data context
- A literal string “`, `”
- The result of a call to a “`substr`” helper with the following parameters:
    - A value of a property “`first_name`” on the current data context
    - A literal integer 0
    - A literal integer 1
- A literal string “.”

Here are some examples of valid templates with expressions from the test cases in `test.js`:

```
<html>
    <head>
        <title>{{ title }}</title>
    </head>
    <body>
    <h1>
        {{ title }}
    </h1>
    <a href="{{ uri }}">Visit</a>
    </body>
</html>
```

```
<html>
    <head>
        <title>{{ title }}</title>
    </head>
    <body>
    <h1>
        {{ title }}
    </h1>
    <a href="{{ createURI 'csep590c' 2016 'Spring' }}">Visit</a>
    </body>
```

```
</html>
```

```
<html>
    <head>
        <title>{{ title }}</title>
    </head>
    <body>
    <h1>
        {{ title }}
    </h1>
    <a href="{{ createURI (concat 'csep' 590 'c') 2016 'Spring'
}}">Visit</a>
    </body>
</html>
```

Task B: Define blocks

A block expression has a *opening part, a body,* and *a closing part*. A body can be an arbitrary combination of raw text, elements, and comments; essentially, it is a full-fledged embedded sub-template. An opening part and a closing part surround the body with a call to a custom *block helper*, which may do arbitrary processing with the template represented by the body.

Opening part

An opening part starts with "`{{#`", ends with "`}}`", and contains *exactly one* application of a block helper in between. An application of a block helper syntactically looks identically to an application of an expr helper (explained above). In particular, it starts with a helper name (which is an identifier), and may contain any number of space-separated parameters (which, by itself, may be literals, data property lookups, or parenthesized subexpressions - even expr helper applications).

Hint: you may want to reuse most of your grammar for parsing expressions to parse block opening parts. If structured properly, you may simply reference parser rules that describe expressions from your parser rule that describes a block opening part.

Closing part

A closing part starts with "`{{/`", ends with "`}}`", and contains exactly one identifier in between. This identifier must be equal to the corresponding block helper name in the opening part.

> **Note:** do not attempt to enforce this requirement in the grammar definition. Instead, you will need to detect an erroneous situation (non-matching opening and closing blocks) in your compiler, and throw an error in this case. Out of the provided test

cases in `test.js`, there is one that verifies that such an error is indeed thrown, and expects its error message to match a certain format.

Problem 2: Compilation (60%)

`HandlebarsCompiler.compile(template)` takes as input a string template and returns an executable JavaScript function. To do that, the compiler does the following:

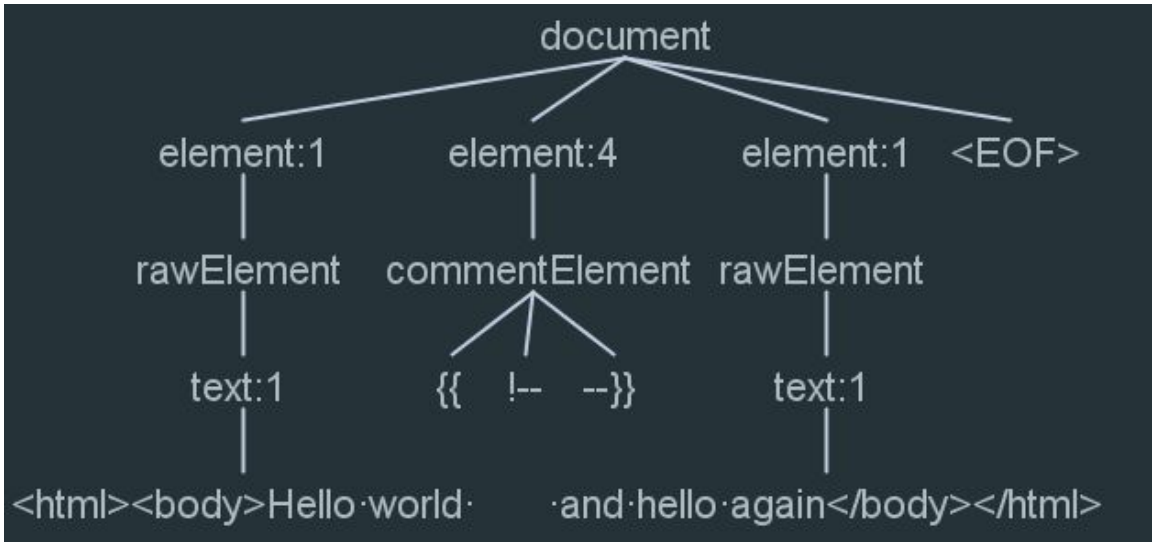
1. Invokes a standard ANTLR pipeline that lexes and parses the template string into an AST.
2. Initiates a tree walk over the constructed AST with itself as a listener.
3. During the walk, it constructs the JavaScript source code of the resulting function in a private field `this._bodySource`.
4. The source code is compiled into an executable function using the standard JavaScript "`new Function`" constructor.

The target function should take as input a *data context* and outputs a string — an instantiation of the compiled template with a given data context. First, this function defines a string variable that will accumulate the resulting text. After that, each instruction in the function appends some text to the result (a raw string or a result of executing a Handlebars expression/block on the data). Finally, it returns the accumulated result.

In the provided skeleton, the input data context variable is called "`__$ctx`" and stored as `this._inputVar.` The output string variable is called "`__$result`" and stored as `this._outputVar.` The method `append` takes as input a string of JavaScript source code and appends this source code to the output variable in the generated function. As an example of its usage, an event handler `exitRawElement` appends a literal string to the output variable — the escaped content of a raw text block represented by the corresponding AST node.

Hint: if you add a `console.log(this._bodySource)` instruction in certain places (e.g. after the tree walk is completed), you can observe the source code of your generated function in the console. You can also use Chrome's built-in step-by-step debugger. Both the console and the debugger are parts of Chrome Developer Tools, which can be invoked by pressing Ctrl+Shift+I (Cmd+Opt+I on OS X).

> **Example 1:** This raw template:
>
> ```
> <html><body>Hello world {{!-- this is a comment --}} and hello
> again</body></html>
> ```
>
> will produce the following parse tree:



and will be compiled into a JavaScript function similar to the following:

```
function (__$ctx) {
      var __$result = "";
      __$result += "<html><body>Hello world ";
      __$result += " and hello again</body></html>";
      return __$result;
}
```

Task A: Compile expressions (without helpers)

Override some event handlers for all expression kinds in your grammar that do not involve calling a helper (that is, for literals, data context lookups, and parenthesized expressions). In these event handlers, construct the expression that should be appended to the result in your generated JavaScript function.

Example 2: This template:

```
<html>
    <head>
        <title>{{ title }}</title>
    </head>
    <body>
    <h1>
        {{ title }}
    </h1>
    <a href="{{ uri }}">Visit</a>
    </body>
```

```
</html>
```

should compile into a function similar to the following:

```
function  (__$ctx) {
      var __$result = "";
      __$result += "\n<html>\n    <head>\n        <title>";
      __$result += __$ctx.title;
      __$result += "</title>\n    </head>\n    <body>\n    <h1>\n
      ";
      __$result += __$ctx.title;
      __$result += "\n    </h1>\n    <a href=\"";
      __$result += __$ctx.uri;
      __$result += "\">Visit</a>\n    </body>\n</html>";
      return __$result;
}
```

Hint: Due to the presence of parenthesized expressions and helpers, your expression grammar is most likely defined recursively. We want to construct the source code for the expression also recursively, and append the constructed expression to the result *only at the top level*. The simplest way to achieve that is to add a *rule context attribute* to your expression rules.

```
expression returns [source]
      : // definition of 'expression'...
      ;
```

For each AST node `ctx` representing a Handlebars expression the `source` attribute will contain the corresponding JavaScript source code of the expression as a field `ctx.source`. You can construct it recursively, assigning `ctx.source` for each possible expression kind in the corresponding "`exit`" event handler (possibly examining the constructed source for children ASTs). At the top level ("`expressionElement`" parser rule), you append the constructed source to the resulting compiled template.

Task B: Add expr helpers

Reminder: an *expr helper* is a user-defined function that can be called during template instantiation. This function has the following signature:

```
function /*name*/ (/*current data context*/ ctx,
                   /*rest of the parameters (named)*/ ...params) {
    // returns a string
}
```

The provided skeleton already contains APIs for registering user-defined helpers. The field `this._helpers.expr` stores a "`string ↦ Function`" dictionary of helper definitions, which

the user populates by calling `compiler.registerExprHelper(name, helper)`. In order to make use of these helpers in your compiled templates, you should:

1. Make sure that these helpers are also defined in the generated JavaScript function.
2. Generate an invocation of a helper with proper parameters when you encounter a *helper application expression*.

Implementing this capability is fairly straightforward, but requires some preparation.

STEP 1: Generate helper definitions.

You want to define helpers in the generated JavaScript function once, before they are called in the compiled template code. After the tree walk is complete, copy all helper definitions from the `this._helpers.expr` dictionary to the generated JavaScript source code as local functions. To do that, use the JavaScript Function.toString() API, which returns the source code of a given function. This code you can prepend to the body of your generated template function.

> **Example 3:** Suppose the user registers the following expr helper:
>
> ```
> compiler.registerExprHelper('eq', function (ctx, value) {
>     return ctx == value;
> });
> ```
>
> (As a reminder, in an expr helper `ctx` is the current data context, and all other parameters (in this case, `value`) are specific to the helper.)
>
> The compiler will store this anonymous function in `this._helpers.expr` under the key `"eq"`. Calling `this._helpers.expr["eq"].toString()` will return its source code:
>
> ```
> "function (ctx, value) {
>     return ctx == value;
> };"
> ```
>
> Now you can locally define `eq` in the generated JavaScript function:
>
> ```
> var eq = function (ctx, value) {
>             return ctx == value;
>          };
> ```

STEP 2: Keep track of used helpers.

We want to define only those helpers in the generated function that are actually referenced in the template we are currently compiling. To achieve that, we should *keep track* of the helpers that are referenced in the template as we are compiling it.

1. Add a new private field `this._usedHelpers` to the compiler (it can be a list of strings, a dictionary, or something more complex). Clear it when the user invokes `compile(template)`.
2. Override some event handler for the parser rule or alternative that represents helper application expressions in your grammar. In this event handler, record the current referenced helper in `this._usedHelpers`.
3. In your code that generates helper definitions after the tree walk is completed, only do so for the helpers that are mentioned in `this._usedHelpers`.

STEP 3: Generate helper invocations.

In the "`exit`" event handler for helper application expressions, construct the JavaScript source code for a helper invocation, and store it in `ctx.source` (similarly to all other expression types). Remember that the first parameter passed into a helper should be the current data context and the rest of the parameters are recursively computed JavaScript expressions.

STEP 4: Mangle helper names.

When you define a helper in the generated JavaScript function as a local variable ("`var helperName = function(...) { ... }`"), you need to make sure that its name does not conflict with any of the JavaScript reserved words, or the function won't compile. If the user defines a helper called "`if`", and you represent it in the generated function as a local variable called "`if`", the call to `new Function` will throw a compilation error.

To fix this, *mangle* the name of a variable representing each helper (e.g. add a prefix to it that will resolve any conflicts). Remember that you need to use the mangled helper name consistently both in its definition and in its usage (invocations that were generated from compiling Handlebars expressions).

> **Example 4:** This template:

```
<html>
    <head>
        <title>{{ title }}</title>
    </head>
    <body>
    <h1>
        {{ title }}
    </h1>
    <a href="{{ makeURI (concat 'csep' 590 'c') 2016 'Spring' }}">Visit</a>
    </body>
</html>
```

> compiled with the following helper definitions:

```
compiler.registerExprHelper('makeURI', function (ctx, course, year, quarter) {
    var quarterId = quarter.slice(0, 2).toLowerCase();
    return `https://${ctx.domain}/${course}/${year%100}${quarterId}`;
});
compiler.registerExprHelper('concat', function (ctx, ...params) {
    return params.join('');
});
```

should generate a function similar to the following:

```
function (__$ctx) {
       var __$concat = function (ctx, ...params) {
            return params.join('');
       };
       var __$makeURI = function (ctx, course, year, quarter) {
            var quarterId = quarter.slice(0, 2).toLowerCase();
            return `https://${ctx.domain}/${course}/${year%100}${quarterId}`;
       };

       var __$result = "";
       __$result += "\n<html>\n    <head>\n        <title>";
       __$result += __$ctx.title;
       __$result += "</title>\n    </head>\n    <body>\n    <h1>\n        ";
       __$result += __$ctx.title;
       __$result += "\n    </h1>\n    <a href=\"";
       __$result += __$makeURI(__$ctx, __$concat(__$ctx, 'csep', 590, 'c'),
       2016, 'Spring');
       __$result += "\">Visit</a>\n    </body>\n</html>";
       return __$result;
}
```

Task C: Add scoping

This task is a preparation for Task D, compiling blocks. Reminder: a *block helper* is a user-defined function that is called during evaluation of a *block element*. It has the following signature:

```
function /*name*/ (/*current data context*/ ctx,
                   /*body template (context → string)*/ body,
                   /*rest of the parameters (named)*/ ...params) {
    // returns a string
    // may call body as a template function with some data context
}
```

The body of a block is an arbitrary collection of template elements — essentially, a sub-template. To invoke a block helper, we need to pass a *compiled function representation of its body* to the helper as a second parameter. This representation should be constructed in the same pass, as we walk over the AST. You already have all the required machinery defined to construct the source code of this function, but it only operates on the top level. We want it to operate on arbitrary level — it should work when you compile a nested block body just as well as it works when compiling top-level expressions. To achieve that, you need to implement *scoping*.

In the new version of the compiler, at each point it will maintain not just the currently generated function code (`this._bodySource`), but a *stack* of currently generated function codes. Every time the compiler encounters a block, it starts a new function scope and pushes it on the stack. It then continues compiling the block body, appending all generated elements to the *function that is currently on top of the stack*. When the compiler exits a block, it pops the generated function code for the block body from the stack. It can now use that code as an argument when calling the block helper.

> **Example 5:** This template:
>
> ```
> <html><body>
> <ul>{{#each episodes}}
>     <li>
>         <h1>{{title}}</h1>
>         {{#if (contains jedi 'Luke')}} Features
> Luke! ⚑{{/if}}
>     </li>
> {{/each}}</ul>
> </body></html>
> ```
>
> has 3 nested scopes. The outermost scope is the entire template surrounding the `{{#each episodes}}` block. The middle scope is the body of the `{{#each episodes}}` block that surrounds the `{{#if}}` block. The innermost scope is the body of the `{{#if}}` block.
>
> As we are compiling the template, the currently generated block on the top of the stack is always the innermost in the chain of nested blocks up to the current location. For instance, at the location marked with ⚑, at the top of the stack you should have a function body similar to the following:
>
> ```
> var __$result = "";
> __$result += " Features Luke! ";
> return __$result;
> ```

The top-level document (the entire template) defines the outermost scope. In the absence of any blocks, this will be the only scope on the stack during compilation. The compiler will append all generated elements to the function code on top the stack, and then, after the tree walk is completed, will pop the function code from the stack and use to generate the final result. So, in the absence of any blocks, the compiler's behavior should be equivalent whether it's represented with a single `this._bodySource` field for a function code or with a stack of such function codes.

1. Change `this._bodySource` to a new field `this._bodyStack`. Initialize it to an empty list at the beginning of `HandlebarsCompiler.prototype.compile(template)`.
   In your code, change all references to `this._bodySource` to instead operate on the string at the top of `this._bodyStack`.
2. Implement two new functions on `HandlebarsCompiler.prototype`: `pushScope()` and `popScope()`.
   In `pushScope`, push a new string to the stack that will serve as a beginning of a nested template function.
   HINT: move initialization of the `__$result` variable in `pushScope`.
   In popScope, finalize the template function at the top of the stack and pop it from the stack.
   HINT: move returning `__$result` to the caller in `popScope`.
3. Change `HandlebarsCompiler.prototype.compile(template)` to use `pushScope` and `popScope`.

Task D: Compile blocks

After the scoping is implemented, compiling blocks is straightforward:

1. Override `enter` and `exit` event handlers on the parser rule or alternative that represents a block body in your grammar.
2. Push a new scope when your enter a block body.
3. Pop a scope when you exit a block. Compile it into a function (using the same `new Function` API). Finally, generate a block helper invocation similarly to an expr helper invocation, passing it the data context, the body function code, and parameter expressions.

   Example 6: This template:

```
<html><body>
<ul>{{#each episodes}}
    <li>{{title}}</li>
{{/each}}</ul>
</body></html>
```

together with the standard definition of the `each` helper (which you will implement in Problem 3), should compile into a function similar to the following (minus the definition of `__$each`):

```
function (__$ctx) {
    var __$result = "";
    __$result += "\n<html><body>\n<ul>";
    __$result += __$each(__$ctx,
                         function (__$ctx) {
                           var __$result = "";
                           __$result += "\n    <li>";
                           __$result += __$ctx.title;
                           __$result += "</li>\n";
                           return __$result;
                         },
                         __$ctx.episodes);
    __$result += "</ul>\n</body></html>";
    return __$result;
}
```

Problem 3: Standard helpers (10%)

Define the following three block helpers and register them in the `HandlebarsCompiler` constructor:

- `{{#each list}}` — takes as input an expression that evaluates to a list, iterates over it, invokes the body template with each list element as a data context, and concatenates the results.
- `{{#if expr}}` — takes as input any expression. If it is truthy, invokes the body template without changing the data context and returns the result; otherwise, returns an empty string.
- `{{#with 'field'}}` — takes as input an expression that evaluates to a string. This string is assumed to be a field name on the current data context. The helper changes the data context to the value of that field, invokes the body template on it, and returns the result.

Deliverables

Please create a new Bitbucket git repository, share it with the csep590c_sp16_overlord account (give us *write* permissions), and tell us its URL in the survey to be posted on Piazza. Your repository should be a clone of the official starter kit, with your additions.

For all problems: your code should extend `HandlebarsLexer.g4`, `HandlebarsParser.g4`, and `compiler.js` in `hw3/`. Your autogenerated ANTLR files (e.g. `HandlebarsParser.js`) will also appear in `hw3/`; you are free to leave them in the repository or ignore them.