ECLIPSE INCUBATION

SeText documentation (Incubation)

Version v0.2

Table of Contents

SeText is a textual syntax specification language and associated scanner/parser generator. It can be used to specify the syntax of a language, and automatically generate a scanner and LALR(1) parser(s).

SeText is one of the tools of the Eclipse ESCET™ project. Visit the project website for downloads, installation instructions, source code, general tool usage information, information on how to contribute, and more.

The Eclipse ESCET project, including the SeText language and toolset, is currently in the Incubation Phase.

For the purpose of this document, it is assumed that the reader is familiar with scanner/parser generators (for example *yacc*/*bison* and *lex*/*flex*), scanner and LALR(1) parser technology (including regular expressions and BNF notation), and understands the limitations of the LALR(1) algorithm.

The following topics are discussed:

1. SeText Reference Manual

This manual explains the SeText language. Topics discussed here are:

1.1. SeText lexical syntax

SeText supports the following comments:

- Everything after `//` until the end of the line is a comment.
- Everything from `/*` up to the next `*/`, possibly spanning multiple lines, is a comment.

SeText keywords may be used by escaping them with a `$` character.

Whitespace (spaces, tabs, and new lines) are essentially ignored.

1.2. Specifying lexical syntax using SeText

Terminals can be specified as follows:

```
@terminals:
  @keywords Operators = and or;
  @keywords Functions = log sin cos tan;
end

@terminals:
  IDTK = "$?[a-zA-Z_][a-zA-Z0-9_]*" {scanID};
end
```

Here we specified two groups of terminals. The first group specifies two keyword (sub-)groups, named `Operators` and `Functions`. For these keywords (`and`, `or`, `log`, etc), terminals are created (`ANDKW`, `ORKW`, `LOGKW`, etc). Furthermore, the keyword group names (`Operators` and `Functions`) may be used as non-terminals in the grammar, to recognize exactly one of the keyword terminals created for that

keyword group.

The second group specifies an `IDTK` terminal, defined by a regular expression (see below). The `{scanID}` part indicates that the resulting tokens should be passed to the `scanID` method in the hooks class (see below), to allow post-processing. Post-processing methods are also allowed for keyword identifiers, such as `sin` and `cos` in the example.

SeText generated scanners use longest match when recognizing tokens. If two or more terminals recognize the same longest match, priorities are used to resolve the conflict. For the example above, the first group of terminals has priority over the second group, thus giving the keywords priority over the identifiers. That is, `@terminals` groups listed earlier in the specification have higher priority than `@terminals` groups listed later in the specification. If two terminals accept the same input, and they are defined within the same group (they have the same priority), then the specification is invalid.

It is also possible to use scanner states:

```
@terminals:
  "//.*";
  "/\*" -> BLOCK_COMMENT;
  @eof;
end

@terminals BLOCK_COMMENT:
  "\*/" ->;
  ".";
  "\n";
end
```

The first group of terminals is for the default state, as no state name is specified. Single line comments (`// ...`) are detected using the first regular expression. This expression is not given a name, and can thus not be used in parser rules.

The second regular expression detects the start of block comments (`/*`) and switches the scanner to the `BLOCK_COMMENT` state.

The second group of terminals is detected only when the scanner is in the `BLOCK_COMMENT` state, as indicated by the `BLOCK_COMMENT` state name after the `@terminals` keyword. Everything except for the end of the comment is ignored (no name for the terminals, and no new scanner state). The end of block comments (`*/`) makes the scanner go back to the default scanner state (arrow without state name).

The `@eof` terminal indicates that end-of-file is allowed in a scanner state (in this case, the default scanner state).

For every scanner, the name of the Java class to generate should be specified, as follows:

```
@scanner some.package.SomeScanner;
```

The scanner class must not be a generic class. Imports (see below) can be used to shorten the specification of the Java class name.

Shortcuts can be used for reuse of regular expressions:

```
@shortcut identifier = "$?[a-zA-Z_][a-zA-Z0-9_]*";

@terminals:
  ID2TK = "{identifier}.{identifier}";
  ID3TK = "{identifier}.{identifier}.{identifier}";
end
```

It is possible to use shortcuts in other shortcuts, as long as a shortcut is defined before its use.

1.3. Regular expressions

Regular expressions are enclosed in double quotes. Within them, the following are supported:

- `a` for character `a`, for any `a` (special characters need escaping).
- `\n` for the new line character (Unicode U+0A).
- `\r` for the carriage return character (Unicode U+0D).
- `\t` for the tab character (Unicode U+09).
- `\a` for character `a`, for any `a` (especially useful for escaping special characters).
- `\\` for character `\` (escaped).
- `\"` for character `"` (escaped).
- `(x)` for regular expression `x` (allows for grouping).
- `xy` for regular expression `x` followed by regular expression `y`.
- `x*` for zero or more times regular expression `x`.
- `x+` for one or more times regular expression `x`.
- `x?` for zero or one times regular expression `x`.
- `.` for any ASCII character except `\n` (new line, Unicode U+0A).
- `x|y` for either regular expression `x` or regular expression `y` (but not both).
- `[abc]` for exactly one of the characters `a`, `b` or `c`.
- `[a-z]` for exactly one of the characters `a`, `b`, ..., or `z`. This notation is called a character class. Note that the ranges of characters are based on their ASCII character codes.
- `[^a]` for any ASCII character except for character `a`. This notation is called a negated character class.
- `{s}` for the regular expression defined by shortcut `s`.

To include special characters, they must always be escaped, wherever they occur in the regular expression. For instance, regular expression `[a\^]` recognizes either character `a` or character `^` (but

not both). Here the ^ character is escaped, as it is a special character (it may be used at the beginning of a character class to invert the character class).

New lines are not allowed in the regular expressions themselves. Obviously, it is possible to detect new lines using regular expressions.

1.4. Terminal descriptions

Terminals can be given an end user readable description (just before the semicolon), for use in parser error messages:

```
@terminals:
  @keywords Operators = and            // "and"
                        or;            // "or"
  IDTK   = "[a-z]+" [an identifier];   // an identifier
  ID2TK  = "[A-Z]+" [ an identifier ]; // an identifier
  ASNGTK = ":=";                       // ":="
  @eof;                                // end-of-file
  X = "[abc]";                         // X
  "[def]";                             // no description
```

Keyword literals (ANDTK and ORTK in the example above) have the keywords surrounded by double quotes as default description. Similarly, terminals defined by regular expressions without choice (no character classes, star operators, etc) and using only 'graphical' characters (no control characters, end-of-file, new lines, etc) also have the literal text that they match (surrounded by double quotes) as default description (see ASGNTK in the example above). The end-of-file token has end-of-file as default description. Keywords that don't have a description and don't have default descriptions as described above, get the name of the terminal as description (see X in the example above). If they don't have a name, they have no description.

Nameless terminals are not used by the parser, and therefore do not require a description. The end-of-file terminal has a default description, and can not be given a custom description. Giving a terminal a custom description if it already has a default description, leads to a warning.

1.5. Imports

Java classes/types can be specified in SeText specifications using their fully quantified names, optionally with generic type parameters:

```
java.util.String
java.util.List
java.util.List<java.util.String>
```

but it is also possible to use imports:

```
@import java.util.String;
@import java.util.String as string;
@import java.util;
@import java.util as u;
```

The first import imports `java.util.String` as `String`. The second imports the same type as `string`. The third import imports the `java.util` package as `util`. The fourth import imports that same package as `u`. After these imports, the following all refer to the `java.util.String` Java type/class:

```
java.util.String
util.String
u.String
String
string
```

It is also possible to import generic types, with their type parameters instantiated:

```
@import java.util.List<java.util.String> as stringList
```

allowing `stringList` to be used as a short form for `java.util.List<java.util.String>`.

Note that it is not possible to use imports to shorten other imports.

Finally, note that Java types where the first part of the identifier (the part before any dot) does not refer to an import, are considered absolute. This means that any Java type name not containing a dot, and not referring to an import, is also considered absolute, and thus refers to a class with that name, in the default package.

1.6. Scanner hooks

As indicated above, the following SeText specification:

```
@terminals:
  IDTK = "$?[a-zA-Z_][a-zA-Z0-9_]*" {scanID};
end
```

defines a terminal `IDTK`, which if recognized, is passed to a `scanID` method for post-processing. If such a call back hook method is specified, a (non-generic) hooks class is required. It can be specified as follows:

```
@hooks some.package.SomeHooks;
```

As for all Java types, imports can be used. For this example, the `some.package.SomeHooks` class must have a default (parameterless) constructor, and an instance method with the following signature:

```
public void scanID(Token token);
```

where the `Token` class is the `org.eclipse.escet.setext.runtime.Token` class. The method may perform in-place modifications to the `text` field of the `token` parameter.

It is allowed to throw `org.eclipse.escet.setext.runtime.exceptions.SyntaxException` exceptions in the hooks methods.

Note that each generated scanner has an inner interface named `Hooks` that defines all the required call back hook methods. The hooks class must implement the interface. This does not apply to scanners that don't have any terminals with call back hooks.

1.7. Specifying grammars using SeText

All SeText grammars start with one or more start symbols:

```
@main  Program    : some.package.ProgramParser;
@start Expression : some.package.ExpressionParser;
```

This specifies two start symbols, the non-terminals `Program` and `Expression`. Each start symbol further specifies the parser class that should be generated for that start symbol. Once again, imports are allowed, and the classes must be non-generic.

There are two types of start symbols:

- regular start symbols (`@start` keyword)
- main start symbols (`@main` keyword)

The main start symbols are exactly the same as the regular ones, except that they must cover the entire grammar. That is, all non-terminals must be reachable from each of the main start symbols. There is no such restriction for regular start symbols.

The non-terminals and rules (or productions) can be specified using a BNF like syntax, as follows:

```
{java.util.List<some.package.SomeClass>}
NonTerm : /* empty */
        | NonTerm2
        | NonTerm NonTerm2
        | NonTerm3 @PLUSTK NonTerm3 SEMICOLTK
        ;
```

This example specifies a non-terminal named `NonTerm`. Once reduced, the call back hooks for this non-terminal must result in a Java object of type `java.util.List<some.package.SomeClass>`. Here, both generic types and imports are allowed.

The non-terminal is defined by four rules (or productions). The first rule is empty, as clarified by

the comment. The comment is obviously not required. The second rule consists of a single non-terminal `NonTerm2`, etc.

Each non-terminal rule gives rise to a call back hook method. The parameters of that method are determined by the symbols that make up that rule. That is, all non-terminal are always passed to the call back hook method. Terminals are only passed to the method if they are prefixed with a `@` character.

1.8. Parser hooks

For parsers, a hooks class must always be specified. The scanner and all parsers share a single (non-generic) hooks class. The following specification (from which we omit the scanner part):

```
@hooks some.package.SomeHooks;

@import some.package.ast;

@main Expression : some.package.ExpressionParser;

{ast.Expression}
Expression : /* empty */
           | Expression @PLUSTK Literal
           | Expression MINUSTK Literal
           ;

{ast.Literal}
Literal : @IDTK
        | PITK
        ;
```

requires a `some.package.SomeHooks` Java class, with a default (parameterless) constructor, and five methods, with the following signatures:

```
public Expression parseExpression1();

public Expression parseExpression2(Expression e1, Token t2, Literal l3);

public Expression parseExpression3(Expression e1, Literal l3);

public Literal parseLiteral1(Token t1);

public Literal parseLiteral2();
```

The return types are determined by the non-terminals. The names of the methods are formed from the text `parse`, the name of the non-terminal, and number of the rule, within the non-terminal. Note that all numbers have equal length. For instance `01`, `02`, `03`, ..., `12`. The parameters consist of all the non-terminals that make up the , as well and those terminals with a `@` before them. The types of the

non-terminal parameters are the types of the corresponding non-terminals. For terminals, the type is the `org.eclipse.escet.setext.runtime.Token` class. The parameter names are formed from their types (first character of the simple name of the class, in lower case), followed by the number of the symbol in the rule, without any `0` prefixes. All numbers start counting at one (`1`).

Note that each generated parser has an inner interface named `Hooks` that defines all the required call back hook methods. The hooks class must implement the interface(s). This interface specifies one additional method, which all parser hooks classes must implement:

```
public void setParser<Parser<?> parser);
```

where the `Parser<?>` class is the `org.eclipse.escet.setext.runtime.Parser` class. This method is provided to allow hooks classes access to the parser that creates the hooks class, and its source information. For more information, see the `getSource` method of the `Parser` class.

An implementation of a hooks class for this example could look like this:

```
package some.package;

import org.eclipse.escet.setext.runtime.Parser;
import org.eclipse.escet.setext.runtime.Token;
import some.package.ast.Expression;
import some.package.ast.Literal;

public class SomeHooks implements ExpressionParser.Hooks {
    @Override
    public void setParser(Parser<?> parser) {
        // No need to store this...
    }

    @Override
    public Expression parseExpression1() {
        return null; // Do something more useful here...
    }

    @Override
    public Expression parseExpression2(Expression e1, Token t2, Literal l3) {
        return null; // Do something more useful here...
    }

    @Override
    public Expression parseExpression3(Expression e1, Literal l3) {
        return null; // Do something more useful here...
    }

    @Override
    public Literal parseLiteral1(Token t1) {
        return null; // Do something more useful here...
    }

    @Override
    public Literal parseLiteral2() {
        return null; // Do something more useful here...
    }
}
```

It is allowed to throw `org.eclipse.escet.setext.runtime.exceptions.SyntaxException` exceptions in the hooks methods. Furthermore, it is allowed to add fold regions to the parser (which then needs to be stored as it is provided via the `setParser` hook method), using the `addFoldRange` methods of the `org.eclipse.escet.setext.runtime.Parser` class.

1.9. Usage hints

Here are some hints on using SeText:

- It is recommended to name the generated and hooks classes, using the following convention:

`XyzScanner`, `XyzParser`, `XyzHooks`, for the scanner, parser, and hooks classes of a language `Xyz` or `XYZ`. For parsers for a part of a language, it is recommended to name the generated parsers `XyzPartParser`, for non-terminal `Part` of language `Xyz` or `XYZ`. Following these naming conventions ensures consistency in the naming of the classes.

- It is recommended to import the packages that contain the classes used as the types of the non-terminals. For instance, import the expressions package `some.long.package.name.expressions` as `expressions` or `exprs`, and then use `{exprs.SomeClass}` as the type for a non-terminal, instead of `{some.long.package.name.expressions.SomeClass}`. Importing the package instead of the individual classes reduces the number of imports, and also avoids conflicts between non-terminals names and class names. For standard Java types, however, it is recommended to import the full type. For instance, import `java.lang.String` or `java.util.List`.

1.10. Generated scanners/parsers

The generated scanners and parsers depend on the `org.eclipse.escet.setext.runtime` and `org.eclipse.escet.common.java` plug-ins. Generated scanners and parsers inherit from the `org.eclipse.escet.setext.runtime.Scanner` class and `org.eclipse.escet.setext.runtime.Parser` class respectively. Look at those classes for the public API of generated scanners/parsers, as it should be fairly self-explanatory.

Besides the scanner and parser(s), debug output is generated from which the scanner and parser(s) can be analyzed. In particular, the debug output for the parsers makes it possible to find out the details about conflicts in the grammar. Furthermore, a skeleton is generated for the hooks class.

2. Using SeText in an Eclipse Plug-in Project

For a new language, follow these steps:

- Create a *Plug-in Project* in Eclipse.
- Add the `org.eclipse.escet.common.java` and `org.eclipse.escet.setext.runtime` plug-ins to the *Required plug-ins* in the project's manifest. Also add any plug-ins that define the classes that you will be referring to in the SeText specification.
- Create the Java package where your scanner, parser(s) and hooks classes are to be stored.
- Create a text file ending with `.setext` in that same package. Fill the specification, and save it.
- Right click the file in the *Project Explorer* or *Package Explorer*, and choose the *Generate Parser(s)* action. Alternatively, right click the text editor for the SeText specification and choose the same action.
- Observe how the files are generated. Make sure the console is free of warnings and errors.
- Copy the hooks class skeleton (extension `.skeleton` to extension `.java`), and implement the hooks.
- You are ready to use the scanner and parser(s).

After changes to the SeText specification:

- Regenerate the code, as before.
- If a `Hooks` interface has changed, update the hooks class.
- You are ready to use the modified scanner and parser(s).

It may be a good idea to put the `.skeleton` file in a version control system. That way, after regeneration, you can ask for a diff. You then know what has changed, and how you need to update the hooks class.

Also note that if a generated `Hooks` interface changes after a regeneration, Java will report errors for methods not yet present in the hook class. Similarly, Java will complain about changed method signatures, and methods that no longer exist in the `Hooks` interface (and thus have invalid `@Override` annotations in the hooks class).

3. Limitations and issues

The following limitations currently apply:

- SeText only allows for the specification of scanners that accept ASCII input.
- SeText currently assumes UTF-8 encoded files. If the input file is actually encoded using a different encoding, scanner exceptions may indicate the wrong character.
- SeText does not support grammars with conflicts (shift/reduce, reduce/reduce, accept/reduce).

4. SeText release notes

The release notes for the releases of SeText and the associated tools, as part of the Eclipse ESCET project, are listed below in reverse chronological order.

The release notes may refer to issues, the details for which can be found at the Eclipse ESCET GitLab issues page.

See also the Eclipse ESCET toolkit release notes covering those aspects that are common to the various Eclipse ESCET tools.

4.1. Version 0.2

Improvements and fixes:

- Documentation has been adapted to be more like the documentation of the other Eclipse ESCET tools (issue #51).

4.2. Version 0.1

The first release of SeText as part of the Eclipse ESCET project. This release is based on the initial contribution by the Eindhoven University of Technology (TU/e).

5. Legal