Skeleton Parser Documentation

Comp 412 – Lab 2

1 Introduction

We provide a simple skeleton parser for testing your LL(1) table generation algorithms. This document describes how to run the parser and the output format that your LL(1) table generation algorithms should produce. The parser we provide is a table-driven LL(1) parser. You will provide the tables that drive the parser. Based on the tables you provide, the parser will recognize strings in the grammar described by the input grammar to your table generator.

1.1 A Note on Lexing

A real parser generator such as YACC would have you specify the interaction between the parser and the lexer. Since we are dealing with a simplified parser generator, we use a simple lexer that splits all tokens on whitespace. The tokens are matched *exactly* with the terminals specified by your table generator.

> **Warning:** Because the lexer splits tokens on white space, white space is significant in YAML. The skeleton parser accepts blanks as valid white space characters, but does not accept tabs.

1.2 Running the Parser

The skeleton parser is written in the Ruby programming language. You must have Ruby installed to run this program. If you are running Mac OS or Linux, chances are you already have Ruby installed. You can download Ruby for free at http://www.ruby-lang.org.

Running the parser is easy. You can ask the parser for help if you forget what to do:

```
$ ruby skeletonParser.rb -h
Usage: skeletonParser.rb -d DEFINITION_FILE [-t] [parser input file]
    -t, --[no-]trace                 Trace Parser Actions
    -d, --definition-file FILE       Parser Table Definitions
```

The parser needs two kinds of input: the table definitions generated from your `llgen` program and the input string you want to parse. Assuming the table definitions are in the file `llgen.yaml` and the input string is in `test.expr`, you would invoke the parser as follows:

```
$ ruby skeletonParser.rb -d llgen.yaml test.expr
SUCCESS
```

The parser prints `SUCCESS` if the string is in the grammar and an error message along with the parser state otherwise. You can use the `-t` option to see a trace of the parser actions as it tries to parse the input string.

If no input file is given, the parser reads from the standard input. This feature allows for easy testing of small strings. Invoke the parser and type your input string. Terminate the input by typing `<CONTROL-D>`.

```
$ ruby skeletonParser.rb -d llgen.yaml
num + num
^D
SUCCESS
```

Recall that the lexer splits all tokens on whitespace, so the tokens in your input string must be separated by whitespace, as shown in the example.

2 LLGEN Output Format

The parser needs a variety of information about the grammar it will parse. Your `llgen` program will output the information needed to drive the parser. To parse a string, the skeleton parser needs to know the terminals, non-terminals, EOF marker used in the parse table, error marker used in the parse table, start symbol, productions, and LL(1) parse table. You must output this information in YAML format.

YAML is a lightweight syntax for exchanging structured information. It is easy for both humans and machines to generate and parse. There are a number of libraries for reading and writing YAML in a variety of programming languages. You are free to use these libraries, but they are not necessary for this assignment as the output format is very simple.

```
terminals: [a, b]
non-terminals: [B]
eof-marker: <EOF>
error-marker: --
start-symbol: B

productions:
  0: {B: [a, B, b]}
  1: {B: []}

table:
  B: {b: 1, a: 0, <EOF>: 1}
```

Figure 1: Required YAML Output for the Grammar $B \Rightarrow a\ B\ b \mid \varepsilon$

2.1 Complete Short Example

An example of the required YAML output for the grammar $B \Rightarrow a\ B\ b \mid \varepsilon$ is shown in Figure 1. Appendix A lists a complete example for the right-recursive expression grammar.

2.2 YAML Quick Intro

The output from your `llgen` program will be represented with maps, lists, maps of lists, and maps of maps in YAML. A map is represented as

```
map1:
    key1: value1
    key2: value2
```

A list is represented as

```
[item1, item2, item3]
```

Maps of lists are represented as

```
mapoflist:
    key1: [item1, item2, item3]
    key2: [item1, item2, item3]
```

Maps of maps are represented as

```
mapofmap:
    key1: {key11: value11, key12: value12}
    key2: {key21: value21, key22: value22}
```

> **Note:** White space is significant in YAML. Map keys must be indented. Additionally, white space (blanks, not tabs) must appear after colons (':') and commas (',').

2.3 Output Details

The output from your `llgen` program will consist of a single YAML map that includes the keys described in the following subsections.

2.3.1 Terminals

The YAML map generated by `llgen` must include a list of all the terminals in the grammar.

key `terminals`

type List

example

```
terminals: [a, b]
```

2.3.2 Non-Terminals

The YAML map generated by `llgen` must include a list of all the non-terminals in the grammar.

key `non-terminals`

type List

example

```
non-terminals: [B]
```

2.3.3 EOF Marker

The YAML map generated by `llgen` must include the EOF value used in the parse table described in § 2.3.7 to represent the END OF FILE coming from the lexer.

key `eof-marker`

type String or Integer

example

```
eof-marker: <EOF>
```

2.3.4 Error Marker

The YAML map generated by `llgen` must include the value used in the parse table to signal a parse error.

key `error-marker`

type String or Integer

example

```
error-marker: --
```

2.3.5 Start Symbol

The YAML map generated by `llgen` must include the start (or goal) symbol for the grammar. The value used for this symbol must occur on the left-hand side of some production in the set of productions described in § 0.

key `start-symbol`

type String or Integer

example

```
start-symbol: B
```

2.3.6 Productions

The YAML map generated by `llgen` must include the set of productions in the grammar. Each production must be keyed with a unique number that is used in the parse table to represent that production. The production itself is represented as a map from a non-terminal to a list of non-terminals and/or terminals.

key `productions`

type Map of (Map of List)

example

```
productions:
  0: {B: [a, B, b]}
  1: {B: []}
```

> **Note:** For productions that have a right-hand side of ε, the non-terminal should map to the empty list, as shown in production labeled 1 in Figure 1 and in the example in this section.

2.3.7 Table

The YAML map generated by `llgen` must include a table that directs the actions of the parser. There must be a row in the table for each non-terminal in the grammar, and a column in the table for each terminal in the grammar and for the EOF marker specified in § 2.3.3.

The table is represented as a map of maps. The outer map represents the rows; it has keys that are the non-terminals specified in § 0. The non-terminals map to another map that has keys that are the terminals specified in § 2.3.1 and the EOF marker specified in § 2.3.3. The value associated with each key in the terminals and EOF marker map is either a production number (§ 0) or the error marker (§ 2.3.4).

key `table`

type Map of (Map of (String or Integer))

example

```
table:
  B: {b: 1, a: 0, <EOF>: 1}
```

3 References

3.1 Ruby

- http://www.ruby-lang.org

3.2 YAML

- http://yaml.org
- http://www.yaml.org/start.html
- http://yaml.org/spec/current.html

Appendix A

Complete Example for the Right-Recursive Expression Grammar

Consider the following right recursive expression grammar:

```
Goal   => Expr                          Term   => Factor Term'
Expr   => Term Expr'                    Term'  => x Factor Term'
Expr'  => + Term Expr'                  Term'  => / Factor Term'
       |  - Term Expr'                  Term'  => ε
       |  ε                            Factor => ( Expr )
                                               | num
                                               | name
```

The skeleton parser accepts the following YAML input for this grammar:

```
terminals: [+, -, x, /, (, ), name, num]
non-terminals: [Goal, Expr, Expr', Term, Term', Factor]
eof-marker: <EOF>
error-marker: --
start-symbol: Goal

productions:
  0:  {Goal: [Expr]}
  1:  {Expr: [Term, Expr']}
  2:  {Expr': [+, Term, Expr']}
  3:  {Expr': [-, Term, Expr']}
  4:  {Expr': []}
  5:  {Term: [Factor, Term']}
  6:  {Term': [x, Factor, Term']}
  7:  {Term': [/, Factor, Term']}
  8:  {Term': []}
  9:  {Factor: [(, Expr, )]}
  10: {Factor: [num]}
  11: {Factor: [name]}

table:
  Goal:    {+: --, -: --, x: --, /: --, (: 0, ): --, name: 0, num: 0, <EOF>: -- }
  Expr:    {+: --, -: --, x: --, /: --, (: 1, ): --, name: 1, num: 1, <EOF>: -- }
  Expr':   {+: 2, -: 3, x: --, /: --, (: --, ): 4, name: --, num: --, <EOF>: 4 }
  Term:    {+: --, -: --, x: --, /: --, (: 5, ): --, name: 5, num: 5, <EOF>: -- }
  Term':   {+: 8, -: 8, x: 6, /: 7, (: --, ): 8, name: --, num: --, <EOF>: 8 }
  Factor:  {+: --, -: --, x: --, /: --, (: 9, ): --, name: 11, num: 10, <EOF>: -- }
```