Zebu Specification

April 15, 2015

1 Overview

Zebu is meant to be a parser generator for the Go language, with it's primary focus being the ability to extend other grammars written in Zebu.

Yacc/Bison, ATNLR, and PPG (Polyglot Parser Generator) serve as inspirations for how Zebu should function, with features pulled from each.

1.1 Why Zebu?

Yacc/Bison laid the groundwork and serve as the model for compiler compilers in the field. Although there are implementations in Go, the syntax is a little outdated and does not feature the main motivation behind Zebu, the ability to extend grammars.

ANTLR serves as a great model for the syntactic design of a modern compiler compiler, but again is missing (and was not designed) the ability to extend grammars. This is not a flaw in ANTLR, but a unique need that requires a specific implementation.

PPG has most of the features that are required to extend a grammar (as that is what it is designed for) but lacks some of the power that is needed to fully extend a grammar.

Zebu is needed to resolve the three above compiler compilers into one that serves a specific purpose, the ability to fully extend grammars.

1.2 Goals

The main goal of Zebu is to generate extensible grammars, as such there will be design tradeoffs versus other compiler compilers.

Zebu highlights...

- Combine lexicon and grammar into one definition similar to ANTLR. The lexicon will be defined with regular definitions. The parser should be defined with a context free grammar.
- Unlike ANTLR, only regular definitions can serve as terminal symbols in the grammar. This is to enforce a consistent style (no mixing regular expressions, regular efinitions, and grammar rules).
- Regular definitions as well as grammar rules will be subject to the same extensible syntax and semantics. This allows extending grammars to override not only the semantics of a grammar, but the syntax as well.
- Generate LL(1) recursive decent parsers that can handle direct and indirect left recursion. The generated parser should be readable by humans.
- No global variables, Zebu should expose a parser object and mechanisms to modify this object. The API Zebu exposes should be operations on this object.
- No parse tree generation. ANTLR parse tree is very useful, but eventually an AST needs to be generated, and thus we have to revert to semantic actions anyways.

1.3 Extention Features

All definitions (regular definitions and grammar rules) are subject to the same extensible syntax and semantics. The expected extention rules are...

- **extend** : Add new productions to an existing production in the grammar.
- **inherit** : Copy and existing production and use it to define a new production. Similar to extend, but the origin production is not modified.
- **override** : Replace a production in the grammar.
- **delete** : Remove a production from a grammar.

2 Zebu Language

2.1 Reserved Words

Zebu reserves the following identifiers for use as keywords.

grammar **import** **keyword** **extend**
inherit **override** **delete**

2.2 Regular Definitions

Regular definitions have the following format and semantic restrictions.

```
Head : Body ;
```

- The head of a regular definitions must begin with an uppercase character and may be followed by zero or more alphanumeric characters. This is done to ease parsing and enforce a style on the actual grammar definition.
- The body of a regular definition may contain any valid unicode character, previously defined regular definition, and regular definition operator.
- The standard dot (.) represents any valid unicode character.
- The regular definition operators are defined as
  - **X* "kleene"** : Match X zero or more times
  - **X+ "plus"** : Match X zero or more times
  - **X|Y "alternation"** : Match X or Y
  - **(X) "group"** : Group X to form an inline regular definition.
  - **X{r} "repeat"** : Match X r times, where r is a range defined as
    - {m} : Match exactly m times
    - {m,n} : Match m through n times
    - {,n} : Match up to n times
    - {m,} : Match at least m times
  - **[?] "class"** : Match any unicode character defined in the character class. Character classes have additional operators.
- The precedence of operators are defined as (highest to lowest)
  - **group**
  - **class**
  - **repeat**
  - **kleene | plus**
  - **alternation**

  - **catenation**

- The regular definition operators for a character class are defined as
  - **[XYZ] "catenation"** : Match X, Y, or Z. Only unicode characters may be used in a class, regular definitions are not allowed.
  - **[^X] "except"** : Match any character not included in the class. Can only be defined at the start of the class.
  - **[X-Y] "range"** : Match any unicode character between X and Y, including both. The range operator **can not** be used on regular definitions.
- The precedence of class operators are defined as (highest to lowest)
  - **except**
  - **range**
  - **catenation**
- Backslash escapes regular definition operators, and is used to access the setup of predefined regular definitions. The set includes
  - \s : Any whitespace character except newline.
  - \n : Newline.

The following is a Zebu grammar describing regular expressions as recognized by Zebu.

```
/* Problem: Clean way to combine regular definition with unicode characters
 * in a way that is easy to parse and use? */

Can we say?

grammar regular_definition;

RegDefStart       : [A-Z] ;
ZebuCharacter     : .
RegDefId          : @(RegDefStart) @(ZebuCharacter)* ;
```

```
TokDot           : \.  ;
TokComma         : ,   ;
TokColon         : \:  ;
TokSemicolon     : \;  ;
TokStar          : \*  ;
TokPlus          : \+  ;
TokBar           : \|  ;
TokCircum        : \^  ;
TokLeftParan     : \(  ;
TokRightParan    : \)  ;
TokLeftBrace     : \{  ;
TokRightBrace    : \}  ;
TokLeftBracket   : \[  ;
TokRightBracket  : \]  ;
TokRange         : -   ;
TokAt            : \@  ;

program
  : regular_definitions
  ;

regular_definitions
  : regular_definition
  | regular_definition regular_definitions
  ;

regular_definition
  : RegDefId TokColon sequence TokSemicolon
  ;

/* Catenation */
sequence
  : /* empty */
  | sequence expression
  ;

expression
  : expression TokBar alternation
```

```
  | alternation
  ;

alternation
  : kleene TokStar
  | kleene TokPlus
  | kleene
  ;

kleene
  : repeat TokLeftBrace repeat_range TokRightBrace
  | repeat
  ;

repeat_range
  : Integer
  | Integer TokComma Integer
  | Integer TokComma
  | TokComma Integer
  ;

repeat
  : TokLeftBracket class TokRightBracket
  | group
  ;

class
  : class_group
  | class class_group
  ;

class_group
  : ZebuCharacter
  | class_range
  ;

class_range
  : ZebuCharacter TokRange ZebuCharacter
```

```
  ;

group
  : TokLeftParan expression TokRightParan
  | word
  ;

word
  : UnicodeCharacter
  | TokAt TokLeftParan RegDefId TokRightParan
  ;
```