Zebu Specification

April 27, 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 Although there are implementations in Go, the syntax is a little outdated and does not feature the main motivation behind Zebu, the ability to extend grammars.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.
- **modify** : Modify the semantic action of a production.

2 Zebu Language

2.1 Reserved Words

Zebu reserves the following identifiers for use as keywords.

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

2.2 Regular Definitions

Regular definitions have the following format and semantic restrictions.

- 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 follows, in order of highest precedence to lowest precedence

group	(X)	Group X to form an inline regular definition.
class	[?]	Match any unicode character defined in the character class.
repeat	X{r}	Match X r times, where r is a repeat range.
kleene	X*	Match X zero or more times.
alternation	X|Y	Match X or Y.
catenation	XY	Match XY.

- A repeat range is defined as follows

{m}	Match exactly m times
{m,n}	Match m through n times
{,n}	Match up to n times
{m,}	Match at least m times

- The regular definition operators for a character class are defined as follows, in order of highest precedence to lowest precedence

except	[ˆX]	Match any character not included in the class. Can only be defined at the start of the class.
range	[X-Y]	Match any unicode character between X and Y, including both.
catenation	[XYZ]	Match X, Y, or Z.

- 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.

```
grammar regular_definition;

EscapeChararacter : \\ .  ;
ZebuCharacter     : @EscapeChararacter | [^\\]  ;

RegDefStart       : [A-Z] ;
RegDefPart        : [a-zA-Z0-9] ;
RegDefId          : @RegDefStart @RegDefPart* ;

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 alternation
  ;

alternation
  : alternation TokBar kleene
  | kleene
  ;

kleene
  : repeat TokStar
  | repeat TokPlus
  | repeat
  ;

repeat
  : class TokLeftBrace repeat_range TokRightBrace
  | class
  ;

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

class
  : TokLeftBracket class_body TokRightBracket
  | group
  ;
```

```
class_body
  : optional_negation class_sequence
  ;

optional_negation
  : /* empty */
  | TokCircum
  ;

class_sequence
  : class_sequence class_word
  | class_word
  ;

class_word
  : ZebuCharacter
  | class_range
  ;

class_range
  : ZebuCharacter TokRange ZebuCharacter
  ;

group
  : TokLeftParan sequence TokRightParan
  | word
  ;

word
  : ZebuCharacter
  | TokAt RegDefId
  ;
```

2.3 Grammar Rules

Grammar rules have the following format and semantic restrictions

- Nonterminals must begin with a lowercase letter, terminals must begin with an uppercase letter and must be a previously defined regular definition.
- Left recursive rules and indirect left recursion are allowed.
- Semanic actions may be defined for each nonterminal. Variables to be used in the semanic actions must be assigned for an element in the rule body. If the RESULT variable is used in the semantic action, a return type must also be specified in the rule head. This explicit syntax makes it easier to follow the semantic actions, verus 1,2 etc.
- All semantic actions have access to an implicit zebu.Parser object.
- Each rule in the RHS of a production can be defined with a label. If a label is not defined, then an implicit label is assigned for each rule in the format "rule#n", where n is the number of the rule in the RHS.
- Any production can be modified using the extention features. The semantics of a modification to a grammar rule are as follows.

extend	Add new productions to a grammar rule.
inherit	Create a new production that inherits all rules of another production.
override	Override and define new rules for a production.
delete	Remove this production from the grammar.
modify	Modify the return type only of a production.

- Specific rules on the RHS can be modified by using the explicit or implic label name. Semantic change slightly when modifying a RHS rule.

extend	RHS rules can not be extended.
inherit	Create a new production using only the RHS rule.
override	RHS rules can not be overriden.
delete	Remove this RHS rule from the production.
modify	Modify the semantic action of the RHS rule.

The following is a Zebu grammar describing grammar rules as recognized by Zebu.

```
grammar grammar_rules;

RegDefStart : [A-Z] ;
RegDefPart  : [a-zA-Z0-9] ;
RegDefId    : @RegDefStart @RegDefPart* ;
```

```
@keyword Extend     : extend    ;
@keyword Inherit    : inherit   ;
@keyword Override   : override  ;
@keyword Delete     : delete    ;
@keyword Modify     : modify    ;

ProdStart   : [a-z] ;
ProdPart    : [a-zA-Z0-9#]  ;
ProdId      : @ProdStart @ProdPart* ;

IdStart     : [a-zA-Z]  ;
IdPart      : [a-zA-Z0-9]  ;
Id          : @IdStart @IdPart* ;

Label       : # ProdId

program
  : productions
  ;

productions
  : /* empty */
  | productions production
  ;

production
  : normal_production
  | extend_production
  | inherit_production
  | override_production
  | delete_production
  | modify_production
  ;

normal_production
  : production_head TokColon production_body TokSemicolon
  ;
```

```
extend_production
  : Extend production_head TokColon production_body TokSemicolon
  ;

inherit_production
  : Inherit TokRightArrow production_head TokColon production_body TokSemicolon
  ;

override_production
  : Override production_head TokColon production_body TokSemicolon
  ;

delete_production
  : Delete ProdId TokSemicolon
  ;

modify_production
  : Modify production_head TokColon optional_action TokSemicolon
  ;

production_head
  : ProdId optional_return_type
  ;

optional_return_type
  : TokEqual Id
  ;

production_body
  : rules
  ;

rules
  : rules TokBar rule
  | rule
  ;
```

```
rule
  : /* empty */ optional_rule_label
  | elements optional_rule_label optional_action
  ;

elements
  : elements element
  | element
  ;

element
  : terminal_or_nonterminal optional_variable_declaration
  ;

terminal_or_nonterminal
  : terminal
  | non_terminal
  ;

terminal
  : RuleId
  ;

non_terminal
  : RegDefId
  ;

optional_variable_declaration
  : /* empty */
  | TokEqual Id
  ;

optional_rule_label
  : /* empty */
  | Label
  ;

optional_action
```

```
  : /* empty */
  | action
  ;

action
  : TokLeftAction code TokRightAction
  ;
```

2.4 The Zebu Object & Metaconstructs

The main interface into the Zebu parser generator is the Zebu object. This is done in order to provide an interface into the semantic actions without having to utilize globals. In addition to a standard set of methods that are available to the programmer, Zebu allows the user to add aditional data and methods to the Zebu object.

- Imports can be added to the Zebu object by utilizing the @imports metaconstruct.
- Fields can be added to the Zebu object by utilizing the @fields metaconstruct. To avoid any name clashes, Zebu fields are prefixed with zb_ and are reserved by the language.
- Methods can be added to the Zebu object by utilizing the @methods metaconstruct. To avoid any name clashes, Zebu methods are prefixed with zb_ and are reserved by the language.
- All semantic actions have an implicit Zebu object that may be referenced. The object is named "zb".

  ```
  lhs=Node
    : rhs
      {:
        zb.buildNode(...)
      :}
    ;
  ```

The following is a Zebu grammar describing the metaconstructs recognized by Zebu.

```
grammar meta_rules;
```

```
Imports : \@imports;
Fields  : \@fields;
Methods : \@methods;

program
  : meta_constructs
  ;

meta_constructs
  : /* empty */
  | meta_construct meta_constructs
  ;

meta_construct
  : meta_import
  | meta_field
  | meta_method
  ;

meta_import
  : Imports TokLeftBrace code TokRightBrace
  ;

meta_field
  : Fields TokLeftBrace code TokRightBrace
  ;

meta_method
  : Methods TokLeftBrace code TokRightBrace
  ;
```