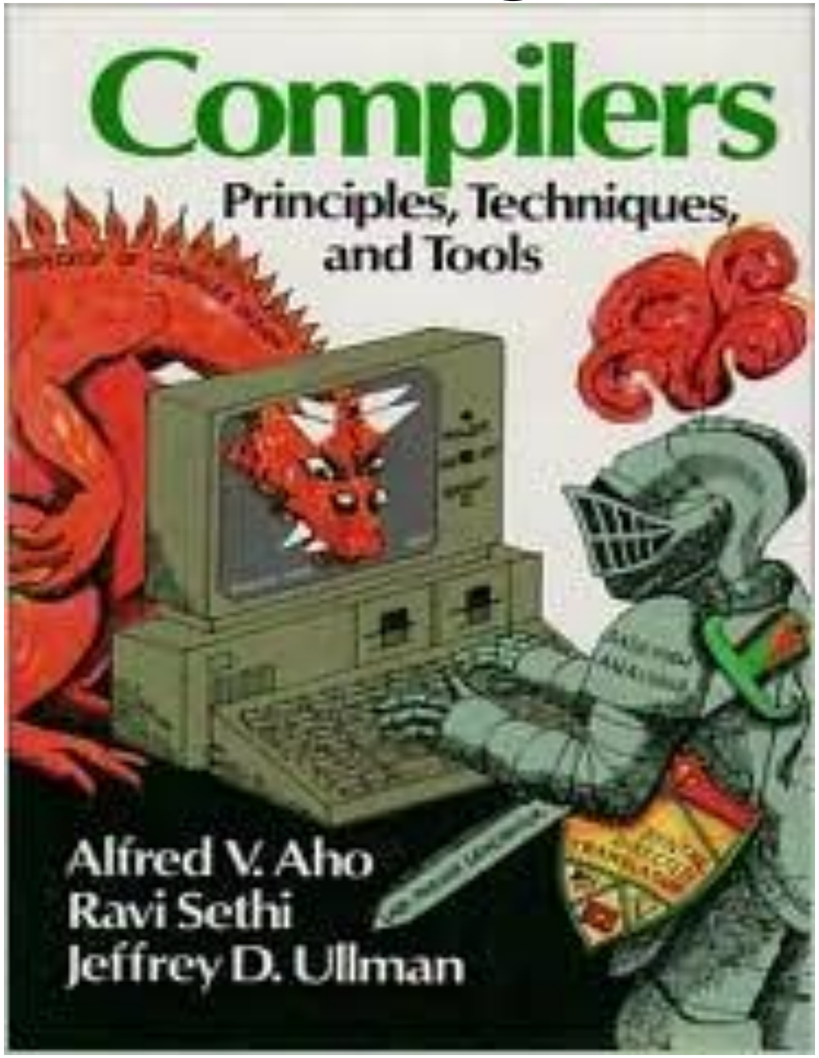
Writing a Compiler using Perl, Pegex and Moo

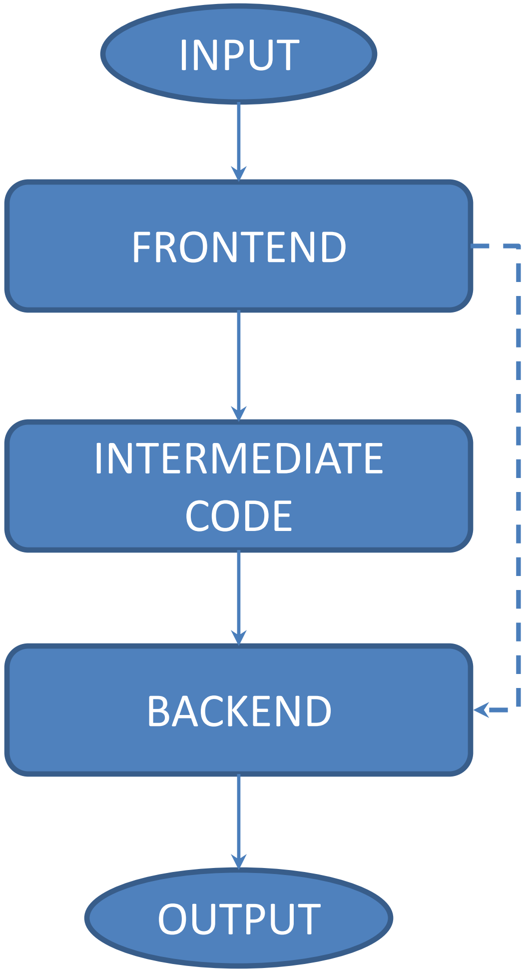
By
Vikas Kumar
vikas@cpan.org
(vicash in #pegex, #vic on freenode)

Compiler Writing Is Difficult !



Terminology

- Frontend
  - Takes text as input
  - Parses it into lexical tokens
- Backend
  - Takes intermediate code or lexical tokens as input
  - Generates target code as output
- Intermediate Code
  - Necessary for compilers with multiple backends and frontends
  - Allows abstraction of backends and frontends
- Abstract Syntax Tree (AST)
  - Allows syntax management
  - Operator precedence management



Compiler Writing Can be Made Easy !

- Use Perl instead of C/C++ to do it
  - Object Oriented Design required for sanity
- Use Pegex to build the Frontend instead of using `bison/yacc` and `flex/lex`.
- Use Moo and Moo::Role to handle multiple Backends
- Use an internal Perl object do handle AST and Intermediate Code (*or not…*)

What remains difficult …

- Optimization of the generated code
  - Avoid if not needed for your case such as for Domain Specific Languages (DSL)
  - Your custom backend may be designed to generate pre-optimized code
- Data flow analysis
  - Can be skipped if not needed, esp. for generating non-assembly language targets like generating Perl/SQL/C++ from a high level DSL
- Verification of Output
  - Absolutely **necessary**, your stuff should work !
  - Tests can only get you so far, you cannot predict your compiler's users.

Overview

- Writing a frontend using Pegex
- Targeting a *single* backend using Pegex::Base
- Targeting *multiple* backends using Moo and Moo::Role
- Example: VIC™ – a DSL compiler for Microchip® PIC® microcontrollers

Writing a Frontend

- Tokenization of input text
  - Traditionally done using a lexer like flex/lex
- Parsing the tokens using a grammar
  - Traditionally done using a grammar generator like bison/yacc
- Create Abstract Syntax Tree (AST)
- Generate intermediate code for the backend

selective intellect

Writing a Frontend

- Tokenization of input text
  - Use Pegex
- Parsing the tokens using a grammar
  - Use Pegex
- Create Abstract Syntax Tree (AST)
  - Use Perl objects or Use Pegex
- Generate intermediate code for the backend
  - Optional: depends on your situation

Pegex Terminology

- Parser: The top level class that is given:
  - The input text to be compiled/parsed
  - The user's Grammar class
  - The user's Receiver class
- Grammar: User provided grammar
- Receiver: A class that has optional functions that:
  - Allow user to handle and modify each token that's parsed
  - Allow user to create AST
  - Allow user to invoke the Backend and generate target code or final output
  - Allow user to create intermediate code, and then call Backend to generate final output

Using Pegex

1. Write a Pegex grammar
   i. Handles both tokenization & parsing at once
   ii. Grammar is similar to writing a Regex
   iii. Greedy parsing will be used
2. Compile the Pegex grammar into a class
   i. Runtime or pre-compiled
   ii. Tree of small regexes used to manage grammar
3. Write a Receiver

Sample Example - VIC™

```
PIC P16F690;
# light up an LED on pin RA0
Main {
   digital_output RA0;
   write RA0, 1;
}
```

```
%grammar vic
program: comment* mcu-select statement* EOS
mcu-select: /’PIC’ BLANK+ (mcu-types | ‘Any’) line-ending/
mcu-types: / ALPHA+ DIGIT+ ALPHA DIGIT+ ALPHA? /
line-ending: /- SEMI – EOL?/
comment: /- HASH ANY* EOL/ | blank-line
blank-line: /- EOL/
statement: comment | instruction | expression | block
# … and so on …
```


selective intellect

Grammar Syntax

- `%grammar <name>`
- `%version <version>`
- `# write comments`
- `<rule>: <combine other rules>`
- The class Pegex::Atoms has a collection of pre-defined rules called ***atoms*** you can use:
  - SEMI (`qr/;/`)
  - EOL (`qr/\r\n|\n/`),
  - ALPHA (`qr/[A-Za-z]/`),
  - DIGIT(`qr/[0-9]/`) and many others.

selective intellect

Using Pegex Grammars

- Save as a `.pgx` file to be compiled using the commandline into a Module
  - Useful for versioned grammars and for release handling
  - Useful for large grammars
- Or use as *string constant* and give to Pegex::Parser for runtime compilation of grammar
  - Useful for small grammars
  - Useful for dynamic grammar class generation if you are into that

selective intellect

Creating Your Grammar Class

```
package VIC::Grammar;
use Pegex::Base;
extends 'Pegex::Grammar';
use constant file => './vic.pgx';
### that's it ###
1;
```

```
$ perl -Ilib -MVIC::Grammar=compile
```

```
package VIC::Grammar;
use Pegex::Base;
extends 'Pegex::Grammar';
use constant file => './vic.pgx';
### autogenerated code ###
sub make_tree {
    {
       '+grammar' => 'vic',
       '+toprule' => 'program',
       '+version' => '0.2.6',
       'comment' => {
           '.any' => [
             {
               '.rgx' =>
                   qr/\G[\ \t]*\r?\n?\#.*\r?\n/
             },
             {
               '.ref' => 'blank_line'
             }
           ]
       },
       #... And so on for other rules ...
    }
}
1;
```

Using Pegex

1. Write a Pegex grammar
   i. Handles both tokenization & parsing at once
   ii. Grammar is similar to writing a Regex
   iii. Greedy parsing will be used
2. Compile the Pegex grammar into a class
   i. Runtime or pre-compiled
   ii. Tree of small regexes used to manage grammar
3. Write a Receiver

Creating Your Receiver Class

- Inherit Pegex::Tree
- For each grammar rule, you **may** write a got_<rule> handler function
- The got_<rule> function:
  - *receives* the parsed token or arrays of arrays of tokens
  - Allows you to modify/ignore the token received
  - Allows you to invoke Backend code if desired
  - Convert the tokens into a custom AST
  - Generate Intermediate Code as needed for the received tokens
- The got_<toprule> or final function can receive complete set of tokens created as array-of-array by Pegex
  - Can be used as an AST as well
  - Return the generated target output from the Backend

```
package VIC::Receiver;
use Pegex::Base;
extends ‘Pegex::Tree’;

has ast => {}; # custom AST object

# single Backend handling
has backend => sub { return VIC::Backend->new; }

# multiple Backend handling.
# Requires got_mcu_type() for the mcu-type rule
has backend => undef;

sub got_mcu_type {
      my $self = shift;
      my $type = shift;
      $self->backend(
                VIC::Backend->new(type => $type));
}

# remove comments from AST
sub got_comment { return; }

# top-rule receiver function
sub got_program {
      my $self = shift;
      my $ast = shift; # use the Pegex generated AST

      print Dumper($ast); # dump the AST if you want

      # … create $output using $self->backend …#
      my $output = $self->backend->generate_code($ast);
      return $output;
}
1;
```

Creating Your Compiler Class

- Create a `Pegex::Parser` object
- Invoke it using your Grammar class and Receiver class
- Provide it input text using the `parse()` function
- Return value is compiled output
- Debugging of the parsing is configurable at runtime

```
package VIC;
use Pegex::Parser;
use VIC::Grammar;
use VIC::Receiver;

sub compile {
    my $input = shift;

    my $parser =
     Pegex::Parser->new(
       ‘grammar’ =>
                  VIC::Grammar->new,
       ‘receiver’ =>
                 VIC::Receiver->new,
        ‘debug’ => 0
     );
     return $parser->parse($input);
}
1;
```


selective intellect

Advantages of Pegex

- Writing Grammars is easy
  - Speed
  - Rapid Prototyping
- No explicit debugging of Regexes required
- Implementing got_<rule> functions will tell you which rule was invoked
- Pegex::Parser with debug set to 1 shows you how the regex matching is done

Writing a Backend

- Needed for code generation for your target
- Example targets:
  - Chips: code generated will be assembly code or binary code
  - Bytecode: JVM/LLVM
    - Write your own Scala/Clojure variant in Perl
  - Code: C/C++/Perl/Lisp/SQL/Lua/Javascript
    - Write high-level logic translators or DSLs

Depending on your Requirements…

Single Backend

- Simpler design
- Target code generation can be done with specialized functions in a single class
- Use Mo/Pegex::Base to keep it light weight, or
- Your Receiver class can have all the code generation functions in it.

Multiple Backends

- Extendable design
- Each target may have some common and some *different* features
- Compiler should handle all the features seamlessly
- Use Moo and Moo::Role for simplicity and extendability

Using Moo::Role with VIC™

- Each chip feature is defined as a Role using `requires`
- Examples:
  - UART
  - USB
  - Timers
- Each feature implementation is also defined as a Role !

```
package VIC::Backend::Roles::Timer
{
      use Moo::Role;
      requires qw(timer_enable
    timer_disable timer_pins);
}
package VIC::Backend::Funcs::Timer
{
    use Moo::Role;
    ## default implementations
    sub timer_enable {
          # … Generate target code …
    }
    sub timer_disable {
          # … Generate target code …
    }
}
```


selective intellect

Using Moo::Role with VIC™

```
package VIC::Backend::P16F690;
use Moo;
use Moo::Role;

# provide custom implementation
sub timer_pins {
   return { TMR0 => [12, ‘TMR’ ] };
}

# inherit the roles and default
   implementations
my @roles = qw(
        VIC::Backend::Roles::Timer
        VIC::Backend::Funcs::Timer
        );
with @roles;
```

```
package VIC::Backend::Roles::Timer
{
     use Moo::Role;
     requires qw(timer_enable
   timer_disable timer_pins);
}
package VIC::Backend::Funcs::Timer
{
   use Moo::Role;
   ## default implementations
   sub timer_enable {
        # … Generate target code …
   }
   sub timer_disable {
        # … Generate target code …
   }
}
```


selective intellect

Checking for a feature

```
package VIC::Backend::Roles::USB
{
     use Moo::Role;
     requires qw(usb_send usb_recv usb_pins);
}
package VIC::Backend::Funcs::USB
{
    use Moo::Role;
    ## default implementations
    sub usb_send {
          my $self = shift;
          # … Give a nice error message here …
          return unless $self->does('VIC::Backend::Roles::USB');
          # … Generate Target Code here …
    }
    sub usb_recv {
          # … Generate Target Code here …
    }
}
```

Using Moo::Role

- Each chip feature is defined as a Role using `requires`
- Examples:
  - UART
  - USB
  - Timers
- Each feature implementation is also defined as a Role with functions
- Functions check if Role is supported for target using `does`

- Allows:
  - Separation of chip details into separate classes
  - Separation of code generation of feature into separate classes
  - Special implementations based on chip internals
  - Compiler can inform user that chip doesn't support a feature in the input code
  - Compiler can list chip features on the commandline

Summary

- Use Pegex to create compiler frontend
  - Writing grammars is like writing a Regex
  - Receiver class contains the main compiler logic
  - Single backend can be in the Receiver class itself
  - Debugging the Grammar is easy
- Use Moo::Role to create multiple backends
  - Allows target feature handling in a clean Object Oriented manner
  - Extendable design
  - Get informative error messages from compiler

Questions ?

https://selectiveintellect.github.io/vic/
Join #pegex on freenode IRC
Join #vic on freenode IRC
Follow us on twitter @selectintellect or @_vicash_