tyrian Documentation

Release 0.1-alpha

Dominic May

January 12, 2014

Contents

Tyrian is a simplistic LISP to python bytecode implementation.

Basic usage is as follows;

```
$ python tyrian.py <options> <input_filename> <output_filename>
```

CHAPTER 1

Contents

1.1 Theory

Subsection contains the theory portion of the project, the parts that do not fit in developer documentation or user documentation.

1.1.1 Documentation

10. Create developer documentation. Annotate your methods in your source code and include this in your developer docs. List each library which is used and why. For external libraries include the website it is available from and the version which you used. If there are any non-standard methods for using or installing the library make sure you document these as well. [3 marks for method doc strings, 3 marks for library usage]

- Methods annotated
- see README.md for access instructions

peak.util.assembler/BytecodeAssembler: this module is used because it removes the need to fully understand the semantics of the python bytecode implementation

- available from http://peak.telecommunity.com/
- docs: http://peak.telecommunity.com/DevCenter/BytecodeAssembler

BytecodeAssembler had been originally written for py2k, and as such, i ported it to py3k.

11. Create user documentation. Describe how to use your product, list your features and what they do, describe known bugs/issues and outline I/O formats. Remember you are communicating with users not developers so use appropriate terminology. [3 marks]

    Right here: *User Documentation*

12. As you progress through this project, keep a developer diary on what you have achieved. This does not need to be a period-by-period account, but when you have complete a milestone, run into (or conquered) a particularly nasty bug or had to redesign a facet of your project because reality didn't mesh with how you envisioned it, write it down. The diary itself is only [2 marks] but will form the basis for your presentation. Don't overlook it!

    This is in the form of *Milestones*

1.1.2 Milestones

Finished milestones

Unknown:

- basic lexer implementation done
- ported BytecodeAssembler from py2k to py3k

Known:

- 02/08 - Got the parser spitting out parse tree's
- 03/08 - Started fleshing out the compiler - disappointingly, this might be easier than I first expected
- 06/08 : 11:01AM - Got code object written to file and executing
- 06/08 : 07:20PM - Got runtime function injection working... :D
- 08/08 - Got basic function definitions working :D
- 11/08 : 11:12PM - Got lisp function arguments working, as far as I can tell
- 14/08 : 07:30PM - Got a basic command line interface implemented
- 21/08 - Documentation now hosted on ReadTheDocs :D
- 22/08 - added entry point for tyrian

Future milestones

- proper scope, needs more testing
- proper lambda's!
- importing?

1.1.3 Project Definition

What is the purpose of your project?

To create a small language, with custom parser and lexer, that compiles into Python Bytecode

What software already exists that either partly solves this problem or can assist you in supporting it?

A few other projects do essentially the same thing as what I am doing.

Ideally, I would:

- Use a third party parser and lexer instead of writing my own.

What are the inputs and outputs of the system?

The inputs are as follows;

- A input filename for a valid lisp program
- An output filename
- Various options

Output:

- A .pyc file

1.1.4 System Development

4. List and describe the minimum features which will need to be implemented for your project to be considered 'successful'. [3 marks]

- Lex, parse, and compile a simple program successfully
- Have comprehensive documentation
- Have a simple program run with correct behavior in the Python VM
- Have basic syntax error detection and notification - full syntax checking does not seem to be possible with a recursive decent parser
- have a nice-ish command line interface

5. How will you evaluate performance of your product? Describe three (3) non-trivial (i.e. not 'program doesn't crash') key performance indicators. [6 marks]

- Does not use excessive amounts of memory
- Runs a simple program within a decent time frame
- compile a program within a decent time frame

6. Create a set of Data Flow Diagrams for your project (at least the Context Diagram and Level 0 DFD - Level 1 if required). [10 marks]

   This is in the form of *Charts*

7. Using your list of features from Part 4, estimate the time it will take to reach them. Create a Gantt or PERT chart for your timeline. Make sure you keep an eye on this, as it will be a significant part of your final presentation. [3 marks]

   See *Assets*

8. Describe at least two areas which could be used to extend your project in future revisions and provide some discussion on what would be required to pursue them. [4 marks]

- Try and have the output be compatible between py3k revisions
- Optimization of the compiled output (this and the one above may be mutually exclusive)
- have it be a full lisp machine, with support for lisp macros (the horror!)

9. Develop and test your project. You must include:

Pseudocode

Pseudocode and a flow chart for one small module of code (must include loops and conditional branching [if statement]) [4 marks]

See the System Development flow chart on the *Charts* page for the flow chart

```
rules <- in rules replace "\n" with " "
rules <- rules split by ";"
rules <- remove excess whitespace from rules
rules <- remove empty rules

while rules do
    rule <- pop from rules

    if rule starts with "%" then
        handle_setting
    else if rule starts with "//" then
        continue
    else then
        value <- rule split by "::="
        key <- first from value

        key <- key uppercased and stripped of excess whitespace

        value <- value joined with "::="

        value <- value cleaned

        loaded_grammars[key] <- value
    end if
end while
```

Test data

Test data for the grammar_parser is available in the tyrian/Grammar folder, test data for the project in general is available in the examples folder.

Trace table

See *Assets* for image

for code

```
rules = rules.replace("\n", " ")
rules = rules.split(";")
rules = map(str.rstrip, rules)
rules = list(filter(bool, rules))

while rules:
    rule = rules.pop(0)

    if rule.startswith("%"):
        handle_setting()
    elif rule.startswith("//"):
        continue
    else:
```

```
        value = rule.split("::=")
        key = value.pop(0)

        key = key.upper().strip()

        value = "::=".join(value)

        value = clean(value)

        loaded_grammars[key] = value
```

Source code

Source code should be included... somewhere.

1.1.5 Hurdles

some problems I had whilst working on this project;

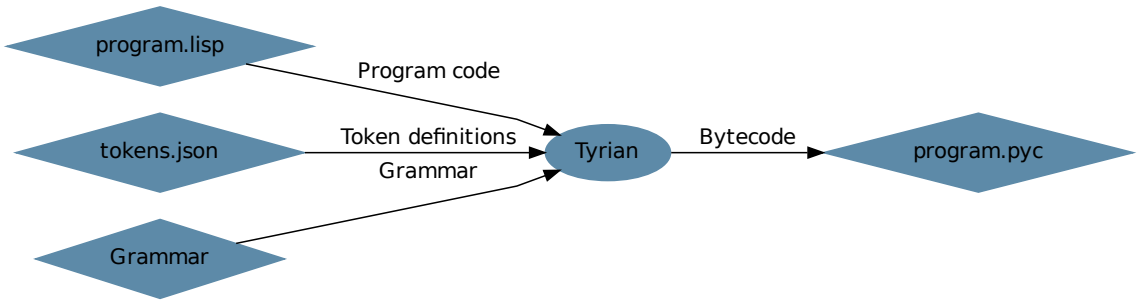
ReadTheDocs;

someone forgot to add a database migration definition file, so I had to contact someone on IRC to fix it

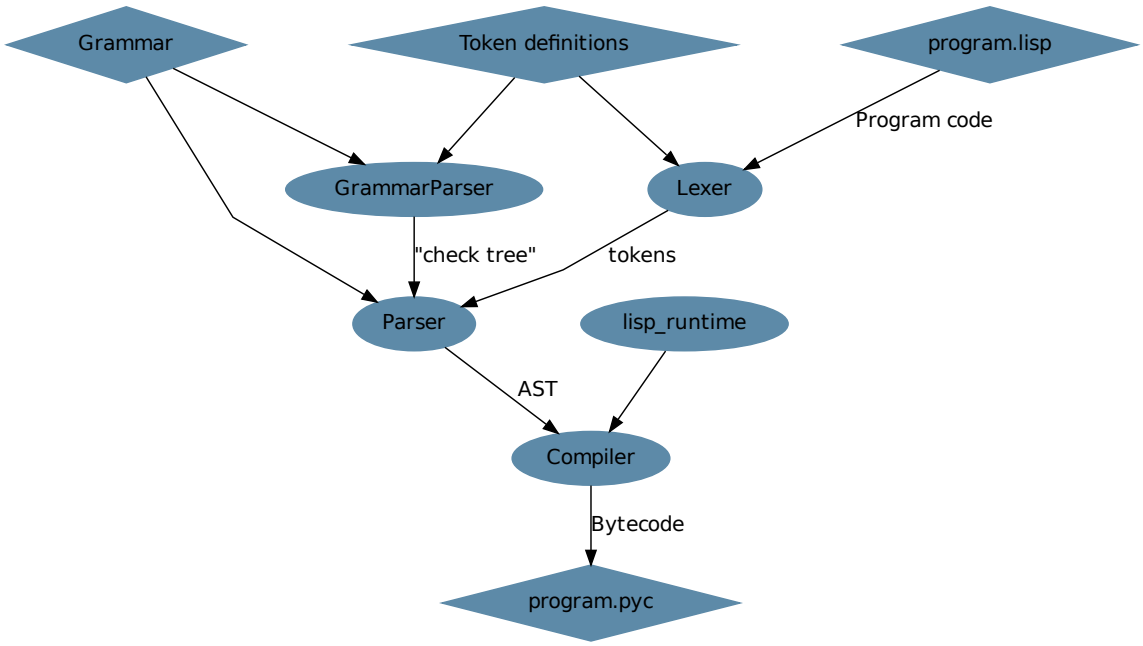
ReadTheDocs uses Python 3.2, and they use by default a version of jinja2 with Unicode literals, which Python 3.2 does not support. As such, I had to build in a virtalenv, and use the requirements.txt file we can provide to force a version of jinja2 that doesn't use Unicode literals, and a version of *markupsafe* that doesn't either. This was corrected a few days later on ReadTheDocs' end.

1.1.6 Charts

Context diagram

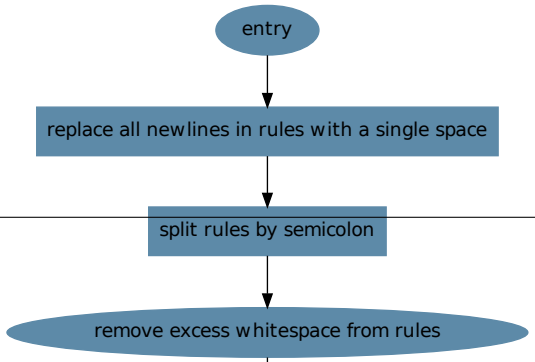


Level 0 DFD



Level 1 DFD

System Development Flow Chart



source files: `../assets/gantt_chart.xlsx`

Trace table

line #	rule	key	value	rules	loaded_grammars	test
1				word ::= words words words; words ::= "blah"		
2				["word ::= words words words;","words ::= 'blah'"]		
3				["word ::= words words words;","words ::= 'blah'"]		
4				["word ::= words words words;","words ::= 'blah'"]		
6						TRUE
7	"word ::= words words words;"			["words ::= 'blah'"]		
9						FALSE
11						FALSE
13			["word ", "words words words;"]			
14		"word "	["words words words;"]			
16		"WORD"				
18			"words words words;"			
20			"words words words;"			
22					key <- value	
6						TRUE
7	"words ::= 'blah'"			[]		
9						FALSE
11						FALSE
13			["words ", " 'blah'"]			
14		"words "	[" 'blah'"]			
16		"WORDS"				
18			" 'blah'"			
20			" 'blah'"			
22					key <- value	
6						FALSE

source files: `../assets/trace_table.py` and `../assets/trace_table.xlsx`

1.2 User Documentation

1.2.1 Quickstart

This article will allow you to get started with Tyrian quickly. Ease is not guaranteed.

```
(print "Hello World")
```

```
$ python tyrian.py hello_world.lisp output.pyc
<debug output>
Writing to file...

$ python output.pyc
Hello World
```

1.2.2 Language Rundown

LISP (LISP Is Syntactically Pure) is what is referred to as a functional language, in that all data manipulation is done via functions;

```
(defvar word "word")

(print word (+ 5 5))

(defun add_five (num)
```

```
    (return (+ num 5))
)

(print (add_five 5))

(let q 5)
(let q (add_five q))

(print q)
```

functions are called via the Polish notation, er, notation.

As this is not intended to be a complete implementation, for ease of implementation, many features have been left out, such as macros. However, the truly core features are present;

- variables
- a standard library
- function calling (of course)
- lisp land function definitions

1.2.3 Standard Library

tyrian comes with a number of standard library functions to assist with programming.

LISP runtime

Contains standard library functions

tyrian.lisp_runtime.registry

tyrian.lisp_runtime.registry.**lisp_function**(***kwargs*)

Registers decorated function in the lisp_registry

if the decorator is being used like so;

```
@lisp_registry
def func():
    pass
```

then we assume the __name__ attribute of the function is to be used

if the decorator is used like so;

```
@lisp_registry(name="blardy")
def randy():
    pass
```

then we use the supplied name :)

tyrian.lisp_runtime.misc

tyrian.lisp_runtime.misc.**call_function**(*func*, **args*)

helper for calling function, usually lambda functions

tyrian.lisp_runtime.misc.**return_func**(*arg*)
when used as the last function call in a function, its output is used as the return value for the function

tyrian.lisp_runtime.simple_math

tyrian.lisp_runtime.simple_math.**sqrt**(*arg*)

tyrian.lisp_runtime.simple_math.**symbol_simple_add**(**args*)

tyrian.lisp_runtime.simple_math.**symbol_simple_div**(**args*)

tyrian.lisp_runtime.simple_math.**symbol_simple_mul**(**args*)

tyrian.lisp_runtime.simple_math.**symbol_simple_sub**(**args*)

1.3 Developer Documentation

1.3.1 tyrian.utils

tyrian.utils.**flatten**(*obj*, *can_return_single: bool=False*)
Flattens nested lists, like so;

```
>>> from tyrian.utils import flatten
>>> flatten([[[[[[[['value']]]]]]]], can_return_single=True)
'value'


>>> flatten([[[[[[[['value']]]]]]]], can_return_single=False)
['value']
```

Parameters
- **obj** – nested list of lists, depth uncertain
- **can_return_single** – see above

tyrian.utils.**enforce_types**(*func: function*)
checks supplied argument types against the annotations
Parametersfunc – function to enforce argument types for

1.3.2 tyrian.lexer

class tyrian.lexer.**Lexer**(*token_defs: dict*)
Code to perform lexing according to token definitions
Parameterstoken_defs – dictionary containing token definitions, see load_token_definitions for definitions

match_with(*left: str*)
Convenience function.

returns an object with a match attribute partial'ed operator.eq, configured to match *left* with the supplied *right*
Parametersleft – const for returned function to be configured to compare against
Return typeobject with *match* attribute

load_token_definitions(*defs: dict*)
Iterates through the supplied token_defs dictionary, creates wrappers for literals and compiles regex's

Parameterstoken_defs – contains token definitions; see `GrammarParser.load_token_definitions` for format

lex(*content: str*, *filename: str*) → list
Takes a string to lex according to token definition loaded via load_token_definitions
Parameters
•**content** – content of file being lexed
•**filename** – name of file being lexed

_lex(*line: str*, *line_no: int*, *filename: str*) → dict
used internally by lex, does actual lexing
Parameters
•**line** – line from source file
•**line_no** – line number of provided line
•**filename** – name of file from which the line originates
yields tokens of format

```
{
    "name": str,
    "token": str,
    "line_no": int,
    'filename': str
}
```

1.3.3 tyrian.nodes

class tyrian.nodes.**AST**(*content*)
Is the overruling object returned from the `Parser`

class tyrian.nodes.**Node**
Base object for Node's

class tyrian.nodes.**ListNode**(*content*, *strip=True*)
Represents a () in LISP

class tyrian.nodes.**ContainerNode**(*content*, *strip=True*)
Aside from being functionally identical to `ListNode`, this Node does not represent anything in the AST, it simply serves as a container; hence the name

class tyrian.nodes.**IDNode**(*content*)
Represents an ID

class tyrian.nodes.**NumberNode**(*content*)
Represents a number

class tyrian.nodes.**StringNode**(*content*)
Represents a string, per se

class tyrian.nodes.**SymbolNode**(*content*)
Represents a mathematical symbol

class tyrian.nodes.**QuotedNode**(**args*, ***kwargs*)
Represents a quoted token

1.3.4 tyrian.tyrian

class tyrian.tyrian.**Tyrian**(*settings:dict=None*)
Primary interface to tyrian
Parameterssettings – dictionary containing settings
compile(*input_filename: str*) → Code
Compile a file into python bytecode
Parametersinput_filename – path to file containing lisp code
Return type Code

1.3.5 tyrian.compiler

class tyrian.compiler.**Compiler**
Handles compilation of AST's

compile_parse_tree(*filename: str*, *parse_tree*) → Code
Takes a filename and a parse_tree and returns a BytecodeAssembler Code object
Parameters
•**filename** – filename of file to compile
•**parse_tree** – parse_tree to compile
Return type Code

write_code_to_file(*codeobject: code*, *filehandler=None*, *filename: str=None*)
Write a code object to the specified filehandler

1.3.6 tyrian.exceptions

exception tyrian.exceptions.**TyrianException**
Base exception to allow for easy catching of all exceptions raised by tyrian

exception tyrian.exceptions.**TyrianSyntaxError**
Raised when a syntax is found

exception tyrian.exceptions.**InvalidToken**
Raised when an invalid token is found

exception tyrian.exceptions.**GrammarDefinitionError**
Raised when the grammar definition file is found to have an error

exception tyrian.exceptions.**NoSuchGrammar**
Raised when a reference grammar does not exist

1.3.7 tyrian.typarser

Contains code for parsing and for building the AST

class tyrian.typarser.**Parser**(***kwargs*)
Simplifies parsing

parse(*lexer:list*) → AST
given a list of tokens, returns a AST
Parameterslexed – list of tokens to parse

1.3.8 tyrian.typarser.grammar_parser

Contains code for parsing the Grammar, and for using it to parse a stream of tokens

grammar_parser.GrammarParser

class tyrian.typarser.grammar_parser.**GrammarParser**(*raw_grammar: dict=None*, *token_defs: dict=None*, *grammar_mapping: dict=None*, *settings: dict=None*)

Does the grunt work of parsing the Grammar into a usable object; see grammar_nodes for more

Parameters

- **raw_grammar** – single string containing raw grammar definitions, see load_grammar
- **token_defs** – dictionary of token definitions, see load_token_definitions

load_grammar_mapping(*nodes*)

Load in a mapping between grammars and Nodes

Supply a dictionary with a mapping between subgrammar names and Node objects

Parametersgrammar_mapping – dictionary mapping subgrammars to appropriate Nodes

load_grammar(*content:str*)

Load grammars from a string. All grammars need not be necessarily be loaded at once, but all must be loaded before parse_grammars() is called.

Parameterscontent – single string containing raw grammar definitions

```
a grammar can be defined like so:
name: <content>;

whereby within the following constructs are permissible;

OR, which can be nested, is denoted by a pipe character:
<token> | <token>

many of a particular token:
<token>+

a subgrammar or token is simply specified by name;
NAME
```

load_token_definitions(*defs: dict*)

Loads token definitions.

expected to be formatted as follows;

Parametersdefs – dictionary containing token definitions

```
{
    'literal': {
        '<content>': '<name>',
        ...
    },
    'regex': {
        '<regex_expr>': '<name>',
        ...
    }
}
```

parse_grammars()
Parses loaded grammars into "check trees".

These "check trees" consist of a root ContainerNode, where a list of tokens can be passed into the root GrammarNode's *check()* function and validated according to the loaded grammars.

parse_grammar(*grammar: str*, *grammar_key: str*, *settings: dict*)
See parse_grammars()

Parameters

- **grammar** – single string containing a single raw grammar definition, see parse_grammars
- **grammar_key** – key for grammar, aka name of grammar
- **settings** – dictionary of settings for Nodes

grammar_parser.grammar_nodes

class tyrian.typarser.grammar_parser.grammar_nodes.**GrammarNode**
Base GrammarNode

class tyrian.typarser.grammar_parser.grammar_nodes.**SubGrammarWrapper**(*settings: dict*, *key: str*, *grammar_parser_inst*) → None
Acts as proxy for subgrammar, ensuring that we need not copy the subgrammar, nor that we need parse the grammars in any particular order.

Parametersgrammar_parser_inst – an instance of the GrammarParser, used to access subgrammars

class tyrian.typarser.grammar_parser.grammar_nodes.**MultiNode**(*settings: dict*, *sub*)
Checks for multiple instances of a set of subnode

Parameterssub – node to checks for multiple instances of

class tyrian.typarser.grammar_parser.grammar_nodes.**LiteralNode**(*settings: dict*, *content*)
Compares a token directly against a string

Parameterscontent – content against which to test

class `LiteralNode`
LiteralNode(content, line_no)

`content`
Alias for field number 0

`line_no`
Alias for field number 1

class `tyrian.typarser.grammar_parser.grammar_nodes.`**`ContainerNode`**(*settings: dict*, *subs: list*) → None
Serves as a container for one or more sub Nodes
Parameterssubs – subnodes to contain

class `tyrian.typarser.grammar_parser.grammar_nodes.`**`RENode`**(*settings: dict*, *regex*, *name*)
Matches a token against a regular expression
Parameters
- **regex** – regular expression to match against
- **name** – name of what the regular expression tests for

class `RENode`
RENode(content, name, line_no)

`content`
Alias for field number 0

`line_no`
Alias for field number 2

`name`
Alias for field number 1

class `tyrian.typarser.grammar_parser.grammar_nodes.`**`ORNode`**(*settings: dict*, *left*, *right*)
checks between two possible sets of subnodes
Parameters
- **left** – node on left side of OR symbol
- **right** – node on right side of OR symbol

t