RULES FEST

Using Rules to Build Languages

Brian Jones

President

Grindwork Corporation

Introduction

- Brian Jones

- 30 years software development
- 15 years running software business
- 13 years technical trainer
- 10 years rapid application development

- Still loving it!

The Dark Ages

- Languages were processed sequentially
  - Single and multi-pass compilers pulled the same source through the process
  - Language designers created language grammars to avoid multi-pass when possible
  - Declarations and definitions required before use
- Why?
  - Limited RAM
  - Slow speed processors
  - Tradition

Tools of the Era

- lex/yacc
  - Classic UNIX C tools
- Antlr
  - Much nicer model
  - Integrates well with JVM and .Net
  - Provides consistent handling for patterns in:
    - Lexing (converting stream of characters to tokens)
    - Parsing (stream of tokens typically to nodes)
    - Tree parsing (pattern-matching for nodes)

Grammars and DSLs

- Fundamentally: parsing is an application of matching and states
  - Grammars are DSLs
  - The define how to recognize and act on patterns in the tokens
- Aren't DSLs good?
  - Yes … when they suit the problem well using them results in less time spent
  - No … when the problem is warped to fit the language instead of the language changed to fit the problem!

Inherent Weaknesses in Grammars

- Linear with limited knowledge
  - Tools process using linear consumption
  - Look-ahead is expensive and sometimes limited
  - Precedence is a primary technique for conflict resolution
- States don't correspond to intention
  - The states may be anonymous in generated state machine (yacc) or recursive-descent position in generated code (Antlr)
  - This makes error reporting tricky

Important: Still great tools!

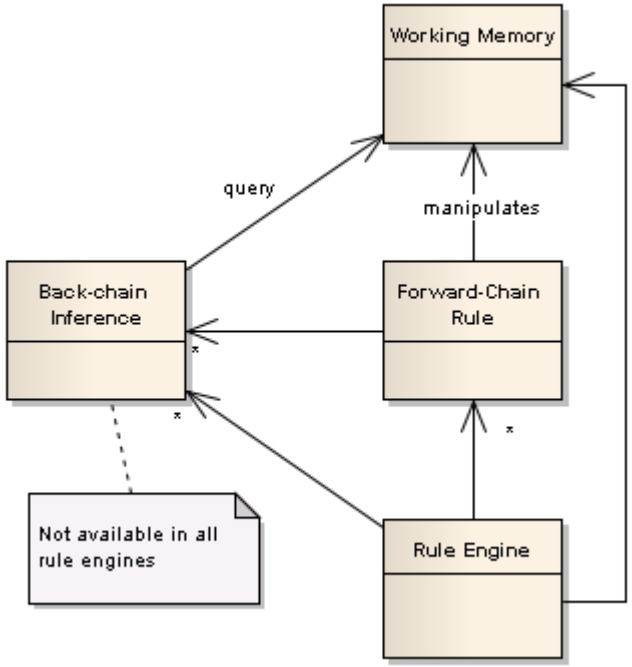
- Yacc and Antlr are still great tools!
  - Often the language you parse is designed with the normal model assumed and these tools work immediately without unusual pain
  - They are well optimized
  - They are well documented
  - They are *relatively* easy to find guidance as needed
- But what happens when they don't fit?
  - That's the rest of the presentation!

Breaking with Tradition

- The grammar desired is a poor match with the grammar DSL (for whatever reason!)
  - Perhaps there's no good DSL for your host language
  - Perhaps there's more dynamics in the language than can be met with a symbol table and simple grouping
- The problem is still matching…
- Rule engines are great at:
  - Matching
  - Dynamic working memory

The Rule Engine Model

- Parsing Equivalences
  - Working memory holds parsed and parsing state
  - Forward-chain rules act as “productions” to add knowledge gained while parsing
  - Back-chains reduce a great deal of repetition



What Happens to the Lexing?

- Lexing isn't discussed here!
- Simple totl-lexer is sufficient for examples
  - Outputs a list of (line-number token) entries
  - Exploded into facts one per token
- What tokens look like:

  (token ?sequence ?line-number ?value)
  Such as:

```
import "somefile.lib" =>
        (token 1 1 IMPORT)
        (token 2 1 "somefile.lib")
```

All the Tokens All the Time

- The rule engine now has 100% knowledge of all tokens in their lexed (sequential) order
  - No need for N-token look-ahead (all tokens available to all rules during entire parse job)
  - No need to process tokens in a single linear pass
  - No need to process tokens in linear order *at all*

Conventional view of tokens limited to sequential processing with single-token look-ahead

All tokens visible in working memory of rule engine during entire job

Processing in Order

- Sequential processing is still trivial using a fact to track current sequence and inferences to access current-token and next-token
  - Fact (current-token-position ?sequence) is asserted (and prior retracted) as the rules process in order
  - Inferences:

```
(current-token ?sequence ?line ?value) <-
        (and (current-token-position ?sequence)
             (token ?sequence ?line ?value))
(next-token ?sequence ?line ?value) <-
        (and (current-token-position ?cur-seq)
             (+ 1 ?cur-seq ?sequence)
             (token ?sequence ?line ?value)
```

Tradeoffs of Sequential Processing

- Benefits
  - Standard grammar productions will map almost verbatim into forward-chain rule definitions
- Downsides
  - Enforces all the sequential processing limits when the environment doesn't have the limits inherently
  - Requires substantially more complex state be stored in working memory to make up for lack of complex matching

Going Beyond Sequential

- Example: Category First Processing
  - Allowing declarations to be foot-noted at the end instead of always put first
- Simple Language
  - Intentionally meant to be processed out of order as a demo!
  - Grammar:
    declare state *state-name*
    declare action *action-name*
    action-name when become *state-name*
  - Every statement must be on a single line
  - ; for comment (thus ;; or ;;;; also comment)

Simple Example

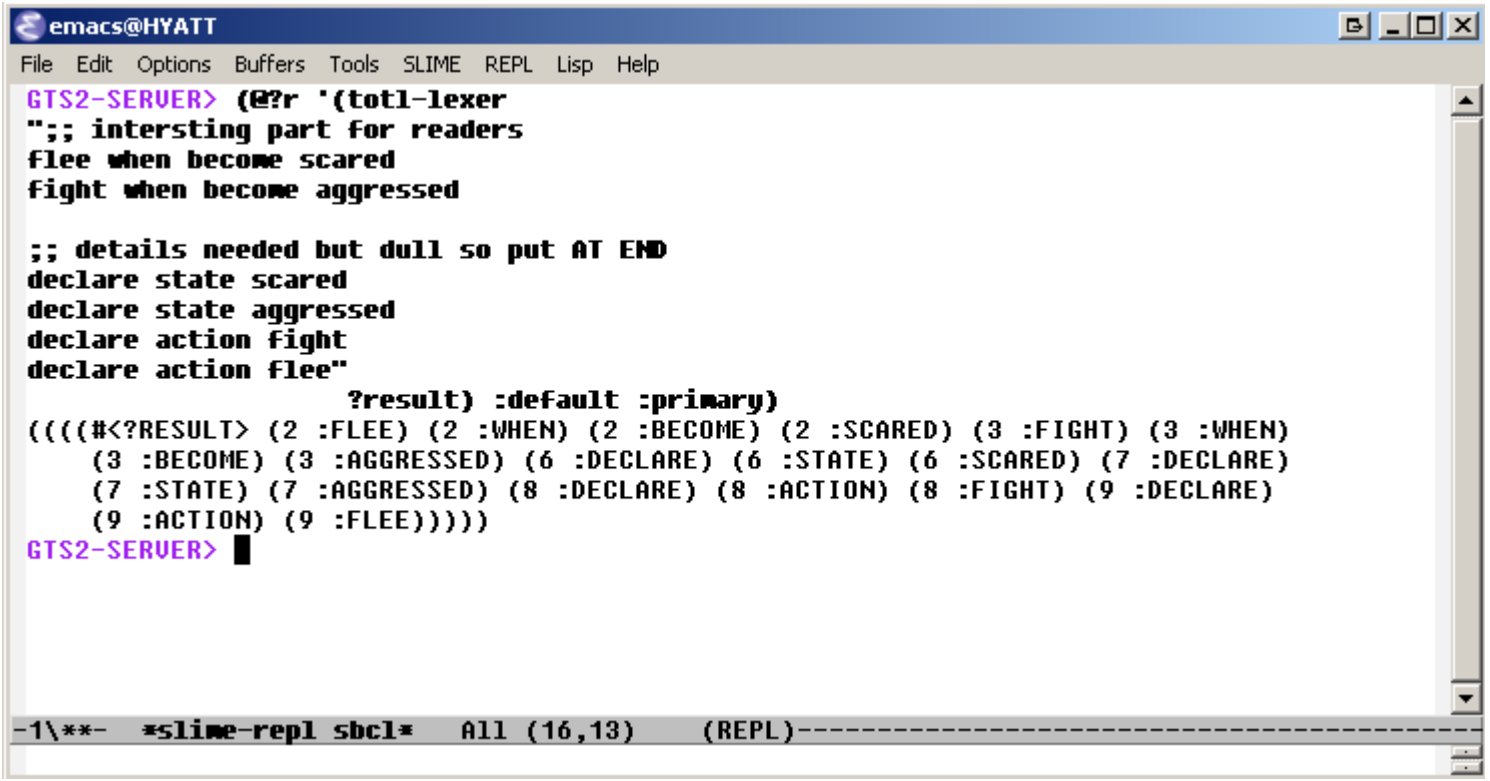
- An example:

```
;; interesting part for readers
flee when become scared
fight when become aggressed

;; details needed but dull so put AT END
declare state scared
declare state aggressed
declare action fight
declare action flee
```

Raw lex result

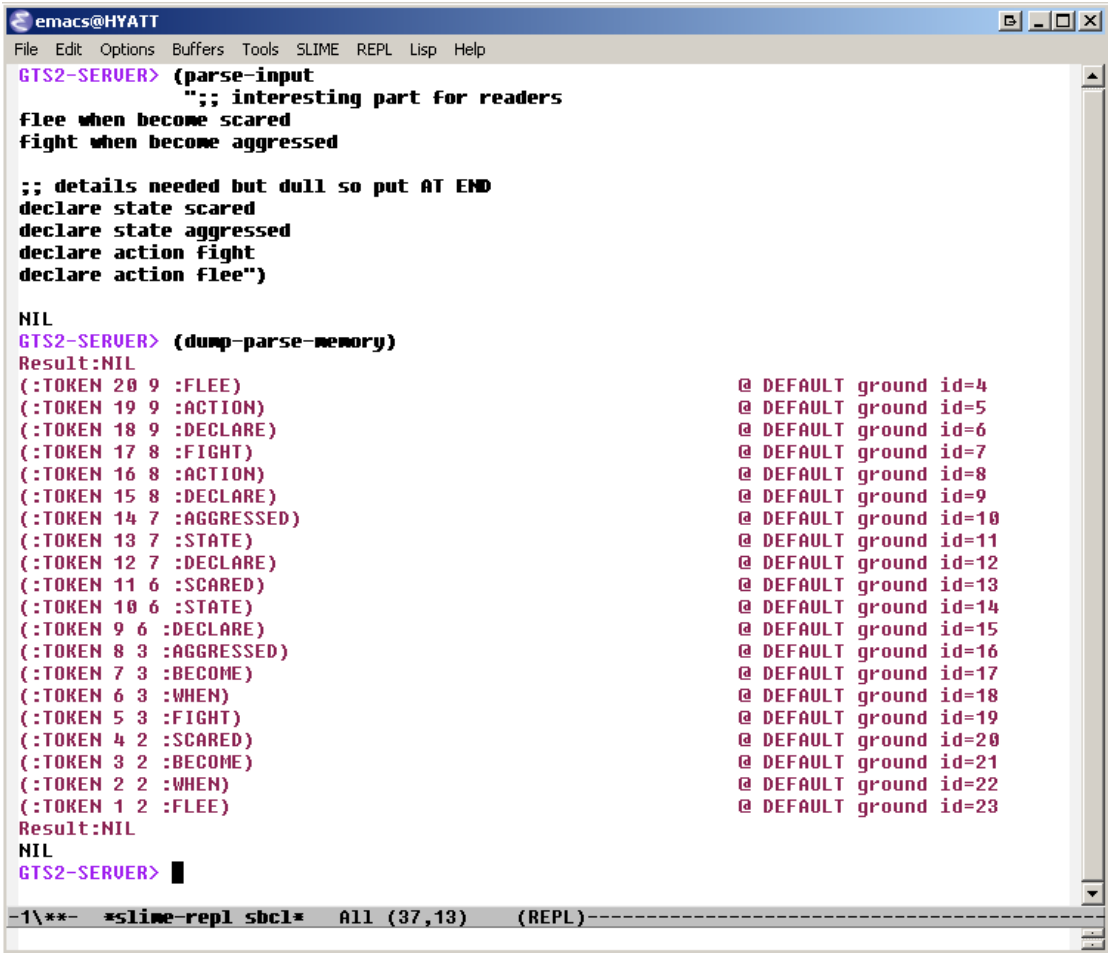
- This is a the result of lexing using the Totl lexer live in emacs:



Lazy Programmer

- Because we're running emacs to a live GTS we have the Lisp REPL ("command line")
- The interactive raw commands (@...) are too much effort when doing a specific task so later screen shots show functions used instead that take less typing
- Interactive consoles to engines are great because they let us interact and simplify as we go

Submitting the Job



No Rules, No work!

- With no rules, nothing happens!

- Why are the tokens reversed in the dump?
  - It happens that the rule that converts the list of raw tokens into token facts reverses the tokens as it adds the sequence count
  - It makes no difference to the results of computation

Adding Rules

- Adding a simple rule to fire when tokens become present gives a clean start:

```
rule recognize-tokens-ready
  when (exists (token . ?))
  rising + (tokens-ready)
end
```

- Trigger emacs with (reload-parse-rules)
- (dump-parse-memory) last entry now shows
  (:tokens-ready)

Playing with Parsing

- Adding rules allows the new rules to fire if they match
- No need to “blow all away” or even re-enter the lexer string
- The interactive nature of the rule engine with command line lets us test and play
- This makes development iteration very fast!

Adding Useful Rules

- Finding valid declarations
  - Rules are needed to match declare <type> <name>
  - Our intent was to find declarations anywhere
  - To avoid processing non-declarations, we'll create a state to match declarations in
- Declarations matching pattern raw and ugly
  - (and (token ?seq ?line declare)
    (+ ?seq 1 ?seq-type)
    (token ?seq-type ?line ?type)
    (member ?type (action state))
    (+ ?seq-type 1 ?seq-name)
    (token ?seq-name ?line ?name))

This is terrible!

- It works … but … it's awful!
- I'd rather do Antlr
- I'd rather do yacc
- I'd rather chew my own leg off!

```
GTS2-SERVER> (@? '(and (token ?seq ?line declare)
                       (+ ?seq 1 ?seq-type)
                       (token ?seq-type ?line ?type)
                       (member ?type (action state))
                       (+ ?seq-type 1 ?seq-name)
                       (token ?seq-name ?line ?name)) :default 29)
---Result 1---
#<?SEQ> = 18
#<?LINE> = 9
#<?SEQ-TYPE> = 19
#<?TYPE> = ACTION
#<?SEQ-NAME> = 20
#<?NAME> = FLEE

---Result 2---
#<?SEQ> = 15
#<?LINE> = 8
#<?SEQ-TYPE> = 16
#<?TYPE> = ACTION
#<?SEQ-NAME> = 17
#<?NAME> = FIGHT

---Result 3---
#<?SEQ> = 12
#<?LINE> = 7
#<?SEQ-TYPE> = 13
#<?TYPE> = STATE
#<?SEQ-NAME> = 14
#<?NAME> = AGGRESSED

---Result 4---
#<?SEQ> = 9
#<?LINE> = 6
#<?SEQ-TYPE> = 10
#<?TYPE> = STATE
#<?SEQ-NAME> = 11
#<?NAME> = SCARED

; No value
GTS2-SERVER>
```

With great power …

- Rule Engines are general purpose
  - They don't "just know" how to convert tokens into useful data
- Teach them!
  - We want a simple abstraction for *line* since our language grammar says that lines are the elements
  - In real world, we normally go with statement, which is slightly (but not shockingly!) harder

Define the usage *first*

- We want to match as easily as antlr/yacc
- We also want to work consistently with our engine
- To match a list (declare state scared) would ideally be as simple as:
  - (match-line ?line-number (declare state ?value))
- Thus, we want the *tool* match-line as follows:
  - (match-line ?line-number ?list-of-values-on-line)

Line-match tool

```
inference

  ;; assemble the line into a simple list that must unify
  (line-match ?number ?list) <-
  (and (line-sequences ?number ?seqs)
       (reverse ?seqs ?seqs-backwards)
       (line-builder ?seqs-backwards ?list ()))

  ;; sequence-numbers-for-line-ordered
  (line-sequences ?number ?sequence-list) <-
  (and (token ?number . ?)
       (subquery (token ?seqs ?number . ?))
       (numeric-sort ?seqs ?sequence-list))

  ;; build the list of values for a line in order
  (line-builder (?next . ?rest) ?result-list-backwards ?so-far) <-
  (and (token ?next ? ?value)
       (line-builder ?rest ?result-list-backwards (?value . ?so-far)))
  (line-builder () ?result-backwards ?result-backwards) <- (true) ;;
(true) prevents from showing in dump

end
```

What does that do?

- Querying using the tool specifying declare:

- GTS2-SERVER> (@? '(line-match ?number (declare ?category ?value)) :default 29)
- ---Result 1---
- (LINE-MATCH 6 (DECLARE STATE SCARED))

- ---Result 2---
- (LINE-MATCH 7 (DECLARE STATE AGGRESSED))

- ---Result 3---
- (LINE-MATCH 8 (DECLARE ACTION FIGHT))

- ---Result 4---
- (LINE-MATCH 9 (DECLARE ACTION FLEE))

- ; No value
- GTS2-SERVER>

Sub-matching (two output styles)

- GTS2-SERVER> (@? '(line-match ?number (?keyword action ?value)) :default 29)
- ---Result 1---
- (LINE-MATCH 8 (DECLARE ACTION FIGHT))

- ---Result 2---
- (LINE-MATCH 9 (DECLARE ACTION FLEE))

- ; No value
- GTS2-SERVER> (@?p '(line-match ?number (?keyword action ?value)) :default 29)
- ---Result 1---
- #<?NUMBER> = 8
- #<?KEYWORD> = DECLARE
- #<?VALUE> = FIGHT

- ---Result 2---
- #<?NUMBER> = 9
- #<?KEYWORD> = DECLARE
- #<?VALUE> = FLEE

- ; No value
- GTS2-SERVER>

Capturing the valid declarations

- So with tools in hand it's time to capture valid declarations first
  - NOTE: this will become MUCH smarter as we iterate!
  - No error checking/reporting/etc. yet

```
rule valid-declarations
  when (and (tokens-ready)
            (not (declarations-captured))
            (line-match ?number (declare ?cat ?value))
            (member ?cat (action state)))
  rising + (declarations-captured)
         + (declared ?cat ?value ?number)
end
```

Result:

```
(:DECLARED :ACTION :FLEE 9)                @ DEFAULT ground id=49
(:DECLARATIONS-CAPTURED)                   @ DEFAULT ground id=50
(:DECLARED :ACTION :FIGHT 8)               @ DEFAULT ground id=53
(:DECLARED :STATE :AGGRESSED 7)            @ DEFAULT ground id=56
(:DECLARED :STATE :SCARED 6)               @ DEFAULT ground id=59
```

But what about errors?

- It's good when valid tokens parse
  - "Testing for success" is a reasonable first step
  - If valid data doesn't parse that's a valid error!
- What can go wrong?
  - Invalid type of declaration (not action or state)
    - This is slightly checked already
  - Too many values after declare (declare action x y…)
  - Too few values after declare (declare action)
- Almost anything else!

The nature of parsing errors

- Nature of error:
  - Not matching a valid line
  - Considering an invalid line valid
  - Not recognizing that a line wasn't processed
  - Not recognizing that lines are internally inconsistent (semantic error!)
- In general:
  - Every line must be known valid or reported as broken
  - Every matched line must be *semantically consistent* with every other matched line
  - Every broken line should *ideally* give some reason

All lines addressed

- Rationale:
  - Parsing is done when every line has been addressed
- Addressing a line
  - A line that is properly handled (such as a declare line generating declared facts) must be recognized as addressed
  - A line that is *identified as broken* must be recognized as addressed
  - Any line not broken or handled is a parser failure and must be reported as such!

Detecting handled lines

- The only handling so far is with the assertion of (declared ?classifier ?identifier ?line-number) facts
- The following inference reports those as handled:
  - (line-handled ?line-number) <-
    (declared ? ? ?line-number)
- Note that a semantic error is still handled
  - It's an additional report later!

Detecting unhandled lines

- A line without a line-handled fact is unhandled/un-addressed
  - (line-not-handled ?line-number) <-
    (and (token ? ?line-number ?)
    (not (line-handled ?line-number)))
- This is a logical assertion if you don't have inferences … do *not* assert ground facts for this or later rules won't cause them to vanish!!

Parsing complete

- How do we know when the parse is done?
  - Some engines *stop* when no rules can fire
  - GTS constantly cycles or can *quiesce* until more assertions/retractions/timers/etc. wake it (which is transparent to rules)
- We setup a rule to detect parse complete
  - Low salience rule so it fires *only when* nothing more can fire
  - Shouldn't fire unless we started parsing (to avoid pre-firing)
  - Should un-fire if parse state changes (such as rules are added to at run-time)

Initial parse-complete detection

```
salience (low -5) (high 10) end

rule detect-parsing-completed
  salience low
  when (and (tokens-ready)
            (exists (line-handled . ?))
            (not (exists (line-not-handled . ?))))
  maintain (parse-completed)
           (parser-stopped)
end

rule detect-internal-parser-error
  salience low
  when (and (tokens-ready)
            (line-not-handled ?line)
            (line-match ?line ?detail))
  maintain (internal-parser-error)
           (parser-stopped)
           (parse-error ?line "Internal parser error line not handled" ?detail)
end
```

End detection results

```
(:TOKENS-READY)                                                    @ DEFAULT ground id=62
(:DECLARED :ACTION :FLEE 9)                                        @ DEFAULT ground id=65
(:DECLARATIONS-CAPTURED)                                           @ DEFAULT ground id=66
(:DECLARED :ACTION :FIGHT 8)                                       @ DEFAULT ground id=69
(:DECLARED :STATE :AGGRESSED 7)                                    @ DEFAULT ground id=72
(:DECLARED :STATE :SCARED 6)                                       @ DEFAULT ground id=75
(:INTERNAL-PARSER-ERROR)                                           @ DEFAULT support=2 id=110
(:PARSE-ERROR 2 "Internal parser error line not handled"
 (:FLEE :WHEN :BECOME :SCARED))                                    @ DEFAULT support=1 id=111
(:PARSER-STOPPED)                                                  @ DEFAULT support=2 id=112
(:PARSE-ERROR 3 "Internal parser error line not handled"
 (:FIGHT :WHEN :BECOME :AGGRESSED))                                @ DEFAULT support=1 id=115
```

Of course – the current data set has errors!

End Detection Facts

- End detection really serves a few purposes
  - Determining when to report to the user the results
  - Determining the validity of the parse
  - Determining the quality of the parser itself
- End markers
  - (parser-stopped) – the job is finished and reporting can be done
  - (parse-completed) – the parser itself worked
  - (internal-parser-error) – the parser failed to handle the input (no user should *ever get this*)

Finishing declarations

- To detect errors in declare lines, we need to know we're done with declare lines
  - This also starts to show the staging (or state-machine)
  - The states are based on domain terms instead of anonymous blocks
  - This ensures that we complete declaration-capture even if there's no declarations

```
rule declares-complete
  salience low
  when (and (tokens-ready)
            (not (declarations-captured)))
  rising + (declarations-captured)
end
```

When are semantic errors checked?

- GTS processes rules in parallel
  - The conflict resolution process provides not a rule to fire but *all legal rules at that instant* to fire
  - Because of this we check issues like name collisions after completed declarations
- In single firing engines
  - The collision detection can be done as part of the identification stage
  - May still be easier after completed to match the legal syntax completely before checking semantics

Example case semantic errors

- Our sample semantic error is the same name declared more than once
- Error fact is:
  - (parse-error ?line-number ?human-string ?any)
  - The ?any allows details to be tacked on that could be used by later error-presentation if needed
  - If the human string is “complete” the ?any could be discarded
  - The simple totl-lexer operator doesn’t provide columnar data or the parse-error would have column as well

First attempt semantic error

- Code has BUG:

```
rule detect-declare-semantic-error
  when (and (declarations-captured)
            (declared ?cat-1 ?name ?number-1)
            (declared ?cat-2 ?name ?number-2))
  calculate (into-string ?string ?name "duplicates line" ?number-2)
  rising + (parse-error ?number-1 ?string (declared ?cat-1 ?name ?number-1))
end
```

The output showing the bug

- Result with same data
  - What went wrong?

```
(:PARSE-ERROR 8 "FIGHT duplicates line 8" (:DECLARED :ACTION :FIGHT 8))    @ DEFAULT ground id=90
(:PARSE-ERROR 9 "FLEE duplicates line 9" (:DECLARED :ACTION :FLEE 9))    @ DEFAULT ground id=93
(:PARSE-ERROR 6 "SCARED duplicates line 6" (:DECLARED :STATE :SCARED 6))    @ DEFAULT ground id=96
(:PARSE-ERROR 7 "AGGRESSED duplicates line 7"
 (:DECLARED :STATE :AGGRESSED 7))                    @ DEFAULT ground id=99
```

Self-duplication

- The problem is that the fact is matching itself!
- To avoid this, ensure that the two matches aren't the same
  - Could match on fact IDs if that's easy in your system (happens to be a nuisance in Totl)

```
rule detect-declare-semantic-error
  when (and (declarations-captured)
            (declared ?cat-1 ?name ?number-1)
            (declared ?cat-2 ?name ?number-2)
            (/= ?number-1 ?number-2))
  calculate (into-string ?string ?name "duplicates line" ?number-2)
  rising + (parse-error ?number-1 ?string (declared ?cat-1 ?name ?number-1))
end
```

All lines addressed – take two

- Update the test data:

```
;; interesting part for readers
flee when become scared
fight when become aggressed

;; details needed but dull so put AT END
declare state scared
declare state aggressed
declare action fight
declare action flee
declare silly bad
```

Results:

(:PARSE-ERROR 10 "Internal parser error line not handled"
 (:DECLARE :SILLY :BAD)) @ DEFAULT support=1 id=94

Making errors better

- The error of the misshapen declare should be declared as an error of declare and not “line not handled.”
- This is a desire to match lines for the purpose of noting *pretty* errors

```
rule detect-misshapen-declare
  when (and (declarations-captured)
            (line-match ?number (declare . ?broken))
            (not (declared ? ? ?number)))
  rising + (parse-error ?number "Declare not shaped properly" (declare . ?broken))
         + (line-handled ?number)
end
```

Catching intentional misshape

- The result shows that the error provided is explicit
  - Creating really good error messages is hard
  - This provides the means to use them if created!
- Fact matching assertion
  - Line-handled is both a fact (here a simple ground) and an assertion
  - This is intentional!

```
(:LINE-HANDLED 10)                                  @ DEFAULT ground id=99
(:PARSE-ERROR 10 "Declare not shaped properly"
 (:DECLARE :SILLY :BAD)))                           @ DEFAULT ground id=100
```

Testing the declare errors

```
;; details needed but dull so put AT END
declare state scared
declare state aggressed
declare action fight
declare action flee
declare state flee
declare silly bad

=➔

(:DECLARED :ACTION :FLEE 9)                                          @ DEFAULT ground id=89
(:DECLARATIONS-CAPTURED)                                             @ DEFAULT ground id=90
(:DECLARED :ACTION :FIGHT 8)                                         @ DEFAULT ground id=93
(:DECLARED :STATE :AGGRESSED 7)                                      @ DEFAULT ground id=96
(:DECLARED :STATE :SCARED 6)                                         @ DEFAULT ground id=99
(:DECLARED :STATE :FLEE 10)                                          @ DEFAULT ground id=102
(:LINE-HANDLED 11)                                                   @ DEFAULT ground id=105
(:PARSE-ERROR 11 "Declare not shaped properly" (:DECLARE :SILLY :BAD))    @ DEFAULT ground id=106
(:PARSE-ERROR 10 "FLEE duplicates line 9" (:DECLARED :STATE :FLEE 10))    @ DEFAULT ground id=109
(:PARSE-ERROR 9 "FLEE duplicates line 10" (:DECLARED :ACTION :FLEE 9))    @ DEFAULT ground id=112
```

Basic Wrap-up

- The actual work is easy because all the types must be known to match
  - (match-line ?line-number (?action when become ?state)
- We also need to capture the knowledge
  - (entry-action ?action ?state ?line-number)
- And recognize that an entry-action fact represents a valid line
  - (line-handled ?line-number) <- (entry-action ? ? ?line-number)

The Rules for the State

```
rule match-valid-actions-on-state-entry
  when (and (declarations-captured)
            (not (entry-actions-matched))
            (line-match ?line-number (?action when become ?state))
             (declared action ?action . ?)
             (declared state ?state . ?))
  rising + (entry-action ?state ?action ?line-number)
         + (entry-actions-matched)
end

inference
  (line-handled ?line-number) <- (entry-action ? ? ?line-number)
end
```

The Entry Action Ready

```
(:ENTRY-ACTION :AGGRESSED :FIGHT 3)                    @ DEFAULT ground id=104
(:ENTRY-ACTIONS-MATCHED)                               @ DEFAULT ground id=105
(:ENTRY-ACTION :SCARED :FLEE 2)                        @ DEFAULT ground id=108
```

And the data is ready to be used!

Lessons Learned

- The DSL in yacc and Antlr make it a lot easier to specify the simple cases!
- The rule engine allows freedom from the linear processing
- The rule engine requires creation of knowledge to “get closer” to the DSL to make the task sensible
- When the rule engine is used adding new features and error handling becomes adding (instead of altering!) more independent rules

Conclusion

- The tiny language complete with tools and error handling (but no code-generation or interpretation – just the parsing!) used roughly 105 lines of code (depending on white-space)
- Eight forward-chain rules were used
- Seven inferences (back-chain rules) were used
- Hope you enjoyed it and enjoy the rest of Rules Fest!