Regular Expression Derivatives in Python

Michael Paddon
mwp@google.com

Motivation

- I want to generate scanners that have guaranteed linear performance and understand Unicode.
- Owens, Reppy and Turon[1] describe how regular expression derivatives may be used to easily convert a regular expression into a deterministic finite automaton. They observe that "RE derivatives have been lost in the sands of time, and few computer scientists are aware of them".

[1] Owens, S., Reppy, J. and Turon, A., 2009.
Regular-expression derivatives re-examined.
Journal of Functional Programming, 19(2), pp.173-190.

Refresher: Regular Expressions

$\emptyset$	null string
ε	empty string
a	symbol in alphabet Σ
$r \cdot s$	concatenation
$r\star$	Kleene closure
$r + s$	logical or (alternation)
$r \,\&\, s$	logical and
$\neg r$	complement

Examples:

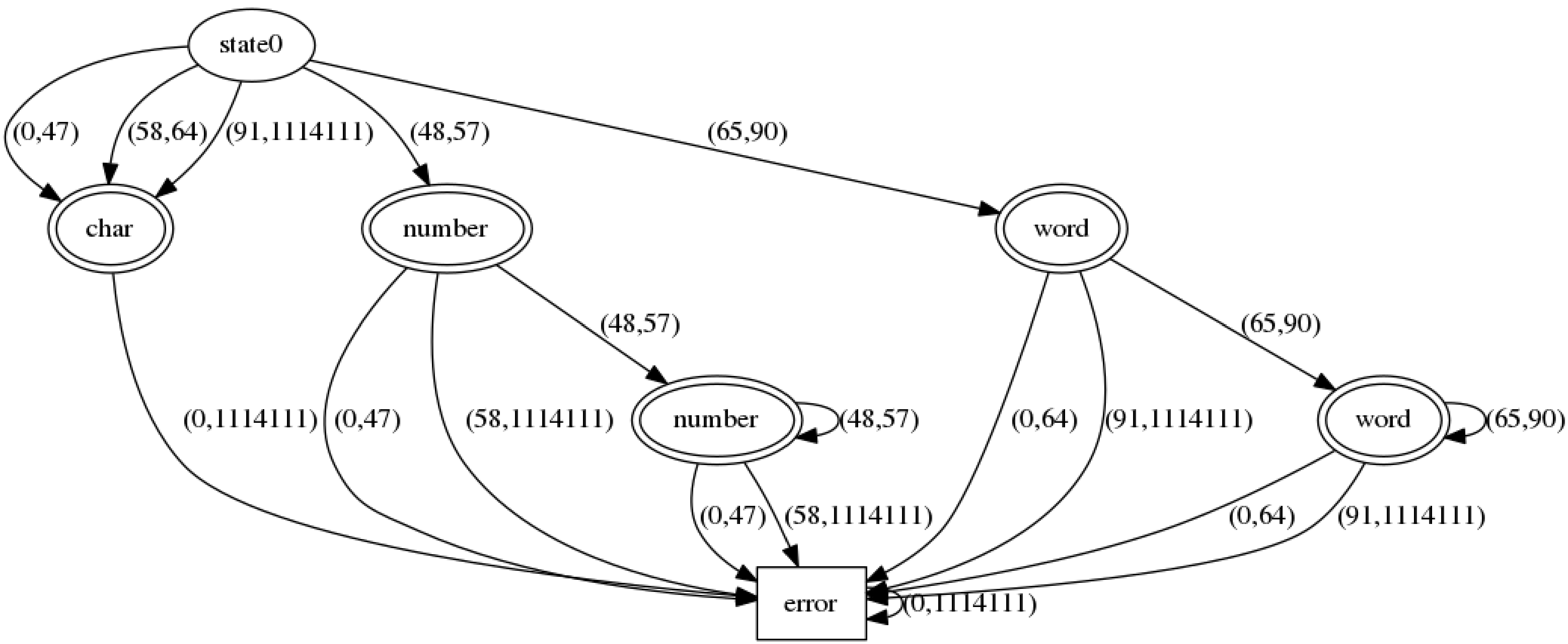
$h \cdot e \cdot l \cdot l \cdot o$

$(a \cdot b \cdot c) + (1 \cdot 2 \cdot 3)$

$a \cdot b^* \cdot c$

Refresher: Deterministic Finite Automata (DFAs)

- Defined as:
  - <states, start, transitions, accepting, error>



Did you know you can take the derivative of a regular expression?

- It's simply what's left after feeding a symbol to an expression…

$$\partial_a a = \varepsilon$$

$$\partial_a b = \varnothing$$

$$\partial_a (a \cdot b) = b$$

$$\partial_a (a^*) = a^*$$

$$\partial_a (a + b) = \partial_a a + \partial_a b = \varepsilon + \varnothing = \varepsilon$$

- This is called a *Brzozowski derivative*.
  - Invented by Janusz Brzozowski in 1964

More generally...

$\partial_a \varnothing = \varnothing$

$\partial_a \varepsilon = \varnothing$

$\partial_a a = \varepsilon$

$\partial_a b = \varnothing$

$\partial_a (r \cdot s) = \partial_a r \cdot s + \nu(r) \cdot \partial_a s$

$\partial_a (r\star) = \partial_a r \cdot r\star$

$\partial_a (r + s) = \partial_a r + \partial_a s$

$\partial_a (r \,\&\, s) = \partial_a r \,\&\, \partial_a s$

$\partial_a (\neg r) = \neg(\partial_a r)$

Helper function ν:

$\nu(\varepsilon) = \varepsilon$
$\nu(a) = \varnothing$
$\nu(\varnothing) = \varnothing$
$\nu(r \cdot s) = \nu(r) \,\&\, \nu(s)$
$\nu(r + s) = \nu(r) + \nu(s)$
$\nu(r\star) = \varepsilon$
$\nu(r \,\&\, s) = \nu(r) \,\&\, \nu(s)$
$\nu(\neg r) = \varepsilon$, if $\nu(r) = \varnothing$
$\nu(\neg r) = \varnothing$, if $\nu(r) = \varepsilon$

If $\nu(r) = \varepsilon$, r is *nullable*

These rules taken from Owens, S., Reppy, J. and Turon, A.,
Regular-expression derivatives re-examined

How is this useful?

```
start = expr
states = {start}
transitions = {start: {}}

stack = [expr]
while stack:
    state = stack.pop()
    for symbol in alphabet:
        next_state = state.derivative(symbol)

        if next_state not in states:
            states.add(state)
            transitions[state] = []
            stack.append(next_state)

        transitions[state].add((symbol, next_state))

accepts = [state for state in states if state.nullable()]
error = states[∅]
```

What about large alphabets?

We can calculate *derivative classes*:

$C() = \{\Sigma\}$
$C(S) = \{S, \Sigma \setminus S\}, S \subseteq \Sigma$
$C(r \cdot s) = C(r)$, if r is not nullable
$C(r \cdot s) = C(r) \wedge C(s)$, if r is nullable
$C(r + s) = C(r) \wedge C(s)$
$C(r \,\&\, s) = C(r) \wedge C(s)$
$C(r*) = C(r)$
$C(\neg r) = C(r)$

For example:
$C(a) = \{a, \Sigma \setminus a\}$

$C(a \cdot b^*) = C(a) = \{a, \Sigma \setminus a\}$

$C(a + b) = C(a) \wedge C(b)$
$= \{a, \Sigma \setminus a\} \wedge \{b, \Sigma \setminus b\}$
$= \{\emptyset, a, b, \Sigma \setminus \{a, b\}\}$

We only need to take a partial derivative for each class instead of each symbol.

These rules taken from Owens, S., Reppy, J. and Turon, A., *Regular-expression derivatives re-examined*

Now we can handle Unicode

```
start = expr
states = {start}
transitions = {start: {}}

stack = [expr]
while stack:
    state = stack.pop()
    for dclass in state.derivative_classes():
        symbol = dclass.any_member_symbol()
        next_state = state.derivative(symbol)

        if next_state not in states:
            states.add(state)
            transitions[state] = []
            stack.append(next_state)

        transitions[state].add((symbol, next_state))

accepts = [state for state in states if state.nullable()]
error = states[∅]
```

Regular Vectors

- We can easily construct a single DFA from a vector of regular expressions!
  - $\partial_a \langle r_1,\ldots,r_n \rangle = \langle \partial_a r_1,\ldots,\partial_a r_n \rangle$
  - $C(r_1,\ldots,r_n) = \bigwedge C(r_i)$
- A sequence of regular expressions, each representing a token, can be reduced to a single DFA.

Vector rules taken from Owens, S., Reppy, J. and Turon, A., *Regular-expression derivatives re-examined*

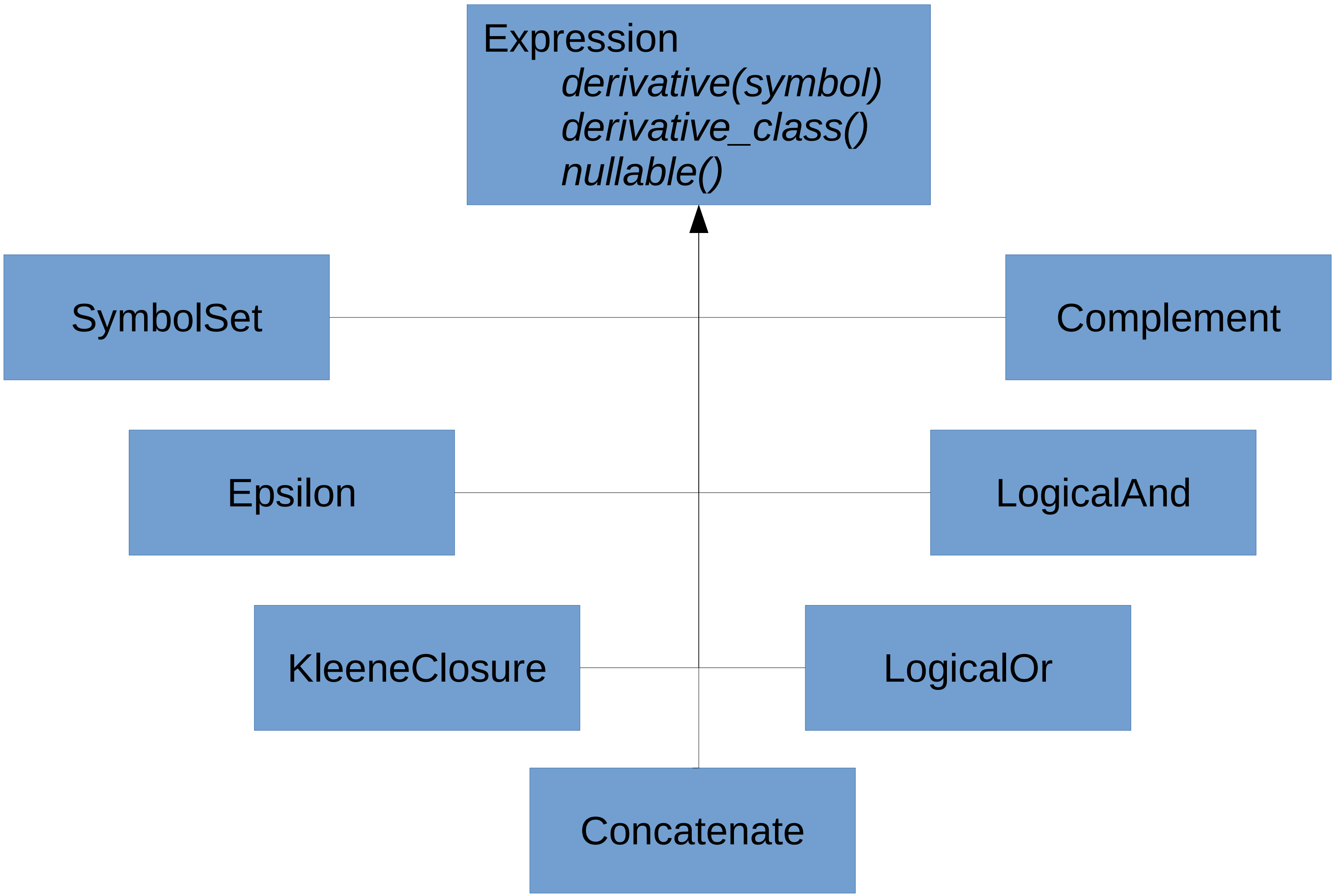
Implementing in Python

- How do we represent large sets of symbols?
- How do we represent expressions?
- How do we compare expressions for equality?
- How do we build a scanner from a DFA?

Large Sets of Symbols

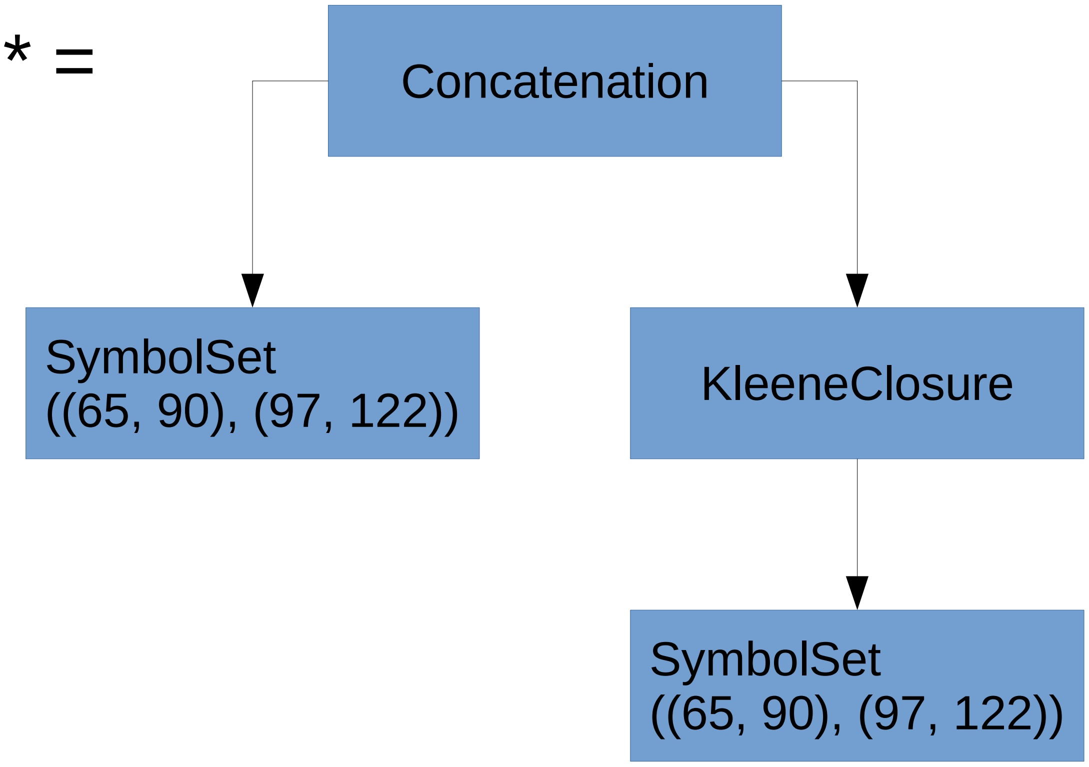
- Represent as ordered disjoint intervals of codepoints
  - e.g. [A-Za-z0-9] → ((48, 57), (65, 90), (97, 122))
  - Testing membership using *bisect() is* O(log N).
  - Union, intersection, difference is O(N)
- Tempting to subclass *collections.abc.Set()* to present a set of integers.
  - But want to support sets of symbol sets → need *hash()*
  - All sets with the same members should hash to the same value
  - The standard hash requires iterating over each member
  - Subclass *tuple* instead with set-like methods.

Expression Class Hierarchy



Expression Trees

[A-Za-z] · [A-Za-z]* =



Expression Equality

- Use *__new__()* as a smart constructor for weak equivalence form which has a total ordering:

$r \,\&\, r \approx r$
$r \,\&\, s \approx s \,\&\, r$
$(r \,\&\, s) \,\&\, t \approx r \,\&\, (s \,\&\, t)$
$\emptyset \,\&\, r \approx \emptyset$
$\neg\emptyset \,\&\, r \approx r$
$(r \cdot s) \cdot t \approx r \cdot (s \cdot t)$
$\emptyset \cdot r \approx \emptyset$
$r \cdot \emptyset \approx \emptyset$
$\varepsilon \cdot r \approx r$
$r \cdot \varepsilon \approx r$

$r + r \approx r$
$r + s \approx s + r$
$(r + s) + t \approx r + (s + t)$
$\neg\emptyset + r \approx \neg\emptyset$
$\emptyset + r \approx r$
$(r^{*})^{*} \approx r^{*}$
$\varepsilon^{*} \approx \varepsilon$
$\emptyset^{*} \approx \varepsilon$
$\neg(\neg r) \approx r$

These rules taken from Owens, S., Reppy, J. and Turon, A.,
Regular-expression derivatives re-examined

Smart Constructor Example

```
class Concatenation(Expression):
    def __new__(cls, left, right):
        if isinstance(left, Concatenation):
            left, right = left._left,
                Concatenation(left._right, right)

        if left == cls.NULL:
            return left
        elif right == cls.NULL:
            return right
        elif left == cls.EPSILON:
            return right
        elif right == cls.EPSILON:
            return left

        self = super().__new__(cls)
        self._left = left
        self._right = right
        return self
```

Building a Scanner

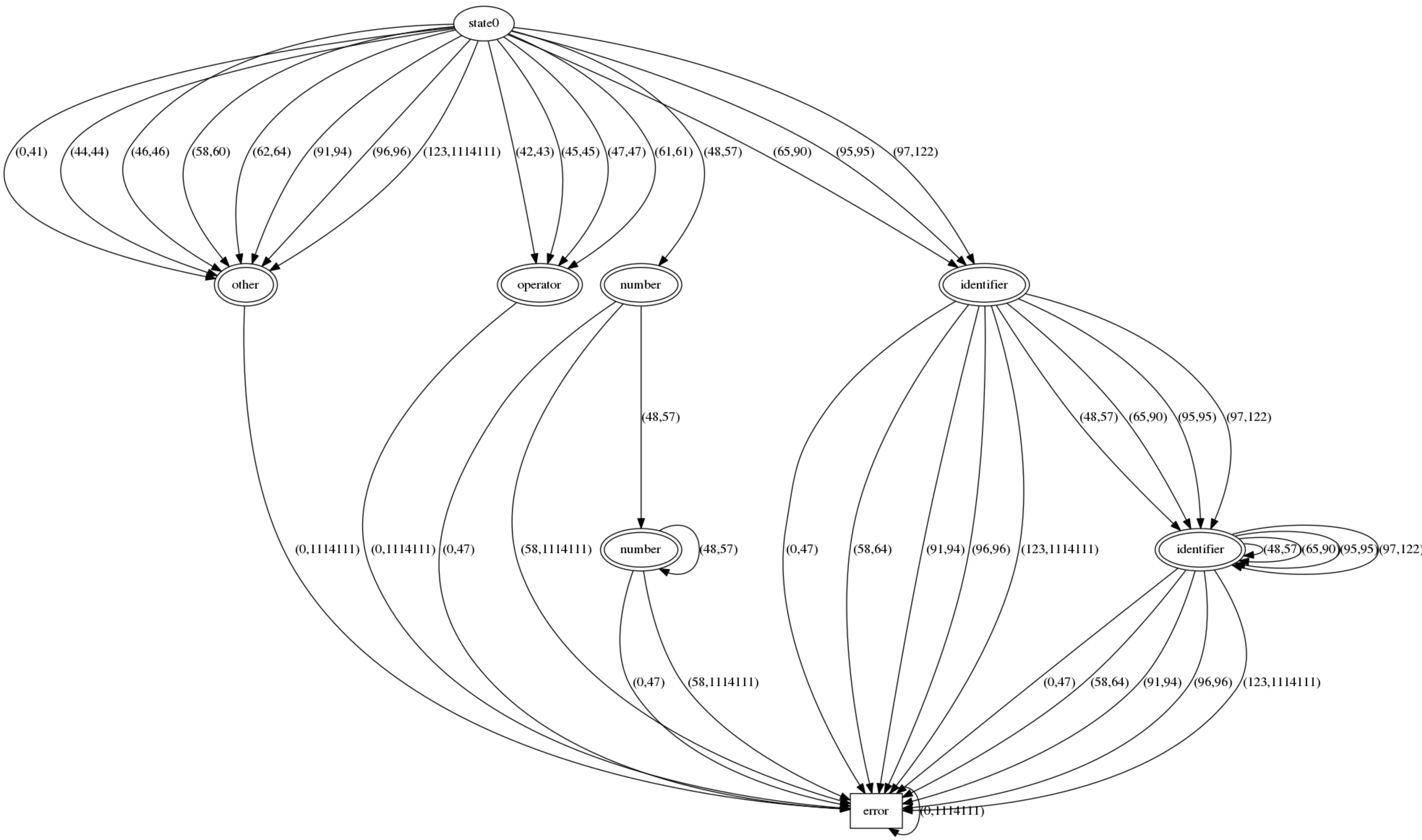
```
state = start
match = None
for symbol in text:
   if state in accepts:
      match = state
      position = current_position()

   state = transition[state][symbol]
   if state == error:
      if match:
         yield match
         rewind_to(position)
         state = start

if match:
   yield match
```

Simple Example

Input looks like a configparser file:

```
[example]
_letter = [_A-Za-z]
_digit = [0-9]
identifier = <_letter>
             (<_letter>|<_digit>)*
number = <_digit>+
operator = [-+*/=]
other = .
```

Resulting DFA



Pascal Lexer

- A larger example:
  - https://github.com/bonzini/flex/blob/master/examples/manual/pascal.lex
  - 51 expressions/tokens
  - flex → 174 states
  - Implemented in epsilon → 169 states

εpsilon

- Supports rich expression syntax:
  - Operators: (), [], !, &, |, ?, *, +, {count}, {min, max}
  - Escapes: mostly perlre compatible, including Unicode classes
- Designed to generate code for multiple targets
  - Currently Python and Dot
- Not done yet:
  - Start conditions, more targets including C
- Code at https://github.com/MichaelPaddon/epsilon
  - Beta testers and contributors welcome!

Acknowledgements

- epsilon was inspired by and directly based on the work of Owens, Reppy, and Turon
- Without the groundbreaking work of Janusz Brzozowski, none of this would be possible.

Thanks!