Generating Tokenizers with Flat Automata

Hans de Nivelle (presenting) and Dina Muktubayeva
Nazarbayev University, Astana, Kazakhstan

Madrid, 21.09.2022

Goals, Motivation

This work is part of a bigger project to design and implement a programming language that is suitable for implementation of logic (proof checking and theorem proving)

Logic is special: Tree-like structures where the types of the subtrees depend on the nodes above it.

Language may be suitable for ASTs too.

As part of this project, we created parser generation tools for C^{++}, taking JFlex (`www.jflex.de/`) and CUP (`www2.cs.tum.edu/projects/cup/`) as example.

(But we are better now.)

I teach compiler construction, and formal languages. I want to be able to show the constructions in class.

Automatic Generation of Tokenizers

There exists a very nice theory for recognizing and classifying tokens:

Regular Expressions ⇒ Non-Determinstic Finite Automata ⇒ Deterministic Finite Automata ⇒ Minimal Deterministic Finite Automaton.

We wanted more flexibility than existing tools. Theory solves 99% of your problems, but you need to allow an escape for this remaining 1%.

Motivation

We introduce a new representation for finite automata as used in tokenizer construction.

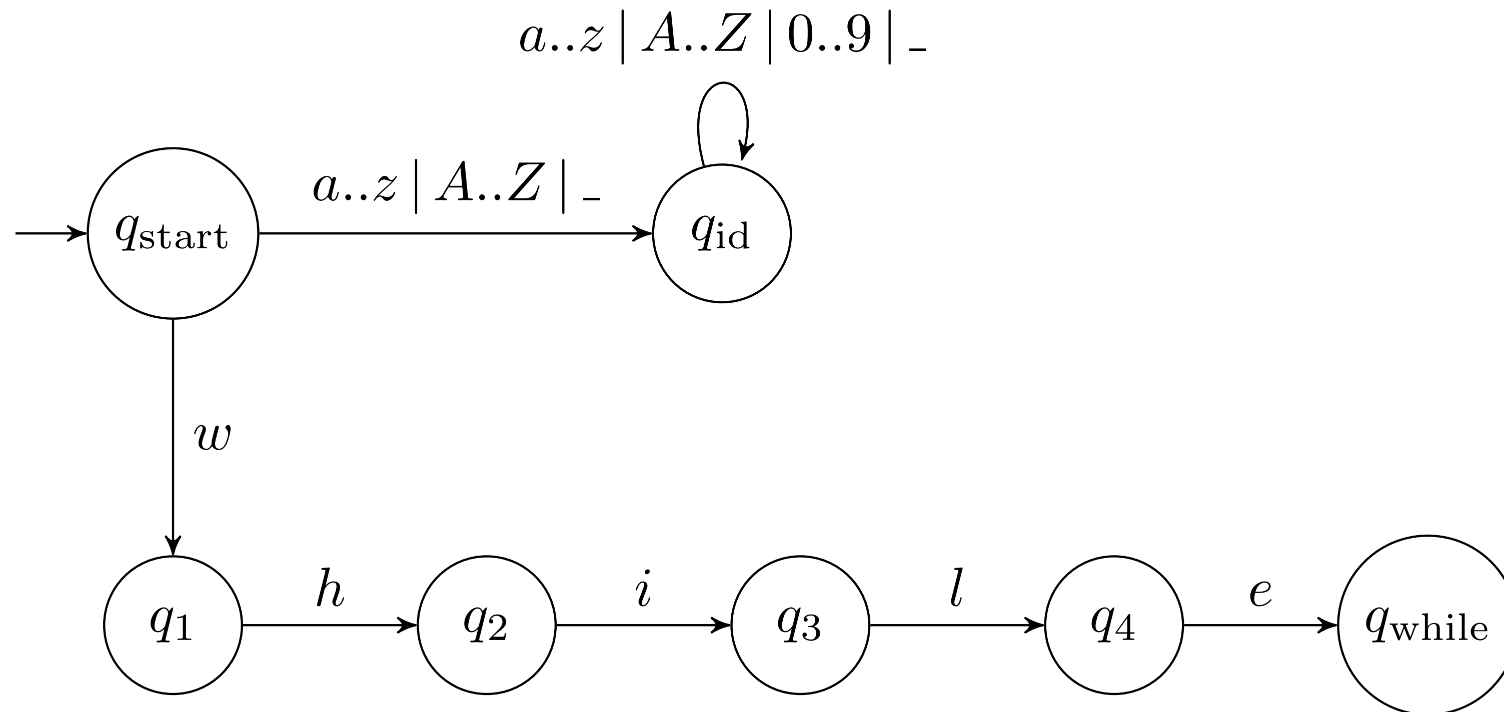
The constructions are slightly more complicated than usually taught, the proofs are slightly more complicated.

What you teach (prove) is what you use.

Implementation is easier.

We have a nice way of dealing with intervals of characters.

Refreshing Automata: Identifier and a reserved Word "while"



Alphabet

An alphabet is a pair $(\Sigma, <)$, s.t. Σ is a non-empty set, and $<$ is a well-order.

We write $c_\perp$ for $\min \Sigma$. We write c^{+1} for the successor of c, if it exists.

Cutting the Intervals

We need to group characters into intervals.

If one wants to deal with big characters sets (like Unicode), one needs an interval-based approach.

Most tokens (with the exception of reserved words) are naturally defined by means of intervals, for example:

$$
\begin{array}{lcl}
\text{digits} & := & [\,'0', \ldots, '9'\,]^{+} \\
\text{exp} & := & ('e' \,|\, 'E')('-' \,|\, '+')^{?}\ \text{digits} \\
\text{doubleconst} & := & ('-')^{?}\ \text{digits}\ \text{exp}\ | \\
 & & ('-')^{?}\ \text{digits}\ ('.')\ \text{digits}^{?}\ \text{exp}^{?}\ | \\
 & & ('-')^{?}\ \text{digits}^{?}\ ('.')\ \text{digits}\ \text{exp}^{?}
\end{array}
$$

Border Functions

Let $(\Sigma, <)$ be an alphabet, let D be an arbitrary, non-empty set. A border function ϕ is a finite sequence of pairs $(\sigma_1, d_1), \ldots, (\sigma_n, d_n)$, s.t.

- $\sigma_1 = \sigma_\perp$,
- For all $1 \leq i < n$, we have $\sigma_i < \sigma_{i+1}$.

We write $\mathrm{dom}(\phi)$ for the set $\{\sigma_1, \ldots, \sigma_n\}$. The set D is called the range of D.

We define the application of ϕ on σ, (we write $\phi^{\leq}(\sigma)$), as follows:

Find the greatest i with $\sigma_i \leq \sigma$. Then $\phi^{\leq}(\sigma) = d_i$.

Minimality

We call a border function $\phi = (\sigma_1, d_1), \ldots, (\sigma_n, d_n)$ minimal if for all $1 \leq i < n$, we have $d_i \neq d_{i+1}$.

If ϕ is not minimal, it can be easily made minimal by removing the redundant pairs. (Even better is to never add them.)

Product

Let ϕ and ψ be border functions over the same alphabet $(\Sigma, <)$.

Let $(\sigma_1, \ldots, \sigma_n) = \text{dom}(\phi) \cup \text{dom}(\psi)$, ordered by $<$

We define the product $\phi \times \psi$ of ϕ and ψ as

$$(\sigma_1, (\phi(\sigma_1), \psi(\sigma_1)\ ), \ldots, (\sigma_n, (\phi(\sigma_n), \psi(\sigma_n)\ ).$$

Application

Let $(\Sigma, <)$ be an alphabet. Let ϕ be a border function with range D. Let f be a function from D to some set D'.

We define the application of f on ϕ, for which we write $f(\phi)$, as follows:

- Write $\phi = (\sigma_1, d_1), \ldots, (\sigma_n, d_n)$.
- Replace it by $(\sigma_1, f(d_1)), \ldots, (\sigma_n, f(d_n))$.
- If the result is not minimal, remove the redundant pairs. (This happens when $f(d_i) = f(d_{i+1})$.)

Perhaps 'composition' would have been a better name.

Acceptors and Classifiers

In order to create a tokenizer, one needs two types of automata:

Acceptors : Accept or reject a word. These are the ones that we teach in formal language class.

Classifiers : Cuts input into pieces and classifies them.

Precisely: Find the longest prefix w' of the input word w, for which there exists a computation towards a state i labeled with a token class that is not **error**.

Among these reachable states i for w', find the most preferred label t_i. Return (w', t_i).

If no such prefix exists, return $(\epsilon, \textbf{error})$.

Acceptors

We define an acceptor as a finite sequence
$\mathcal{A} = (\Lambda_1, \phi_1), \ldots, (\Lambda_n, \phi_n)$, where each $\Lambda_i \in \mathcal{Z}$, and each ϕ_i is a border function from $(\Sigma, <)$ to $\mathcal{Z} \cup \{\#\}$.

Each Λ_i denotes the set of epsilon transitions from state i, while each ϕ_i represents the set of 'consuming' transitions from state i.

- The initial state is always 1, and the accepting state is always $n + 1$.
- State references in Λ_i or ϕ_i use relative addressing.
- $\#$ means that no transition is possible.
- There are no transitions to state < 2 or $> n + 1$. (Note that the first part implies that the acceptor is committing)

nr	Λ	ϕ
$1:$	$\{1\}$	$\{\,(c_\perp, \#)\,\}$
$2:$	$\{2\}$	$\{\,(c_\perp, \#), (a, 1), (z^{+1}, \#)\,\}$
$3:$	$\{2\}$	$\{\,(c_\perp, \#)\,\}$
$4:$	$\{\}$	$\{\,(c_\perp, \#), (A, 1), (Z^{+1}, \#)\,\}$
$5:$	$\{1\}$	$\{\,(c_\perp, \#)\,\}$
$6:$	$\{2, 4, 6, 8\}$	$\{\,(c_\perp, \#), (a, 1), (z^{+1}, \#)\,\}$
$7:$	$\{2\}$	$\{\,(c_\perp, \#)\,\}$
$8:$	$\{\}$	$\{\,(c_\perp, \#), (A, 1), (Z^{+1}, \#)\,\}$
$9:$	$\{2\}$	$\{\,(c_\perp, \#)\,\}$
$10:$	$\{\}$	$\{\,(c_\perp, \#), (0, 1), (9^{+1}, \#)\,\}$
$11:$	$\{2\}$	$\{\,(c_\perp, \#)\,\}$
$12:$	$\{\}$	$\{\,(c_\perp, \#), (_, 1), (_^{+1}, \#)\,\}$
$13:$	$\{-7\}$	$\{\,(c_\perp, \#)\,\}$

Constructing Acceptors

Writing border functions by hand is not trivial, because we don't want to know the order of characters.

Let $(\Sigma, <)$ be an alphabet.

Define

$$\begin{aligned}
\phi_\emptyset &= \{\, (\sigma_\perp, \mathbf{f}) \,\} \\
\phi_\Sigma &= \{\, (\sigma_\perp, \mathbf{t}) \,\}
\end{aligned}$$

$$\begin{aligned}
\phi_{\geq\sigma} &= \text{ if } (\sigma = \sigma_\perp) \text{ then } \{\, (\sigma_\perp, \mathbf{t}) \,\} \text{ else } \{\, (\sigma_\perp, \mathbf{f}), (\sigma, \mathbf{t}) \,\} \\
\phi_{\leq\sigma} &= \text{ if exists}(\sigma^{+1}) \text{ then } \{\, (\sigma_\perp, \mathbf{t}), (\sigma^{+1}, \mathbf{f}) \,\} \text{ else } \{\, (\sigma_\perp, \mathbf{t}) \,\}
\end{aligned}$$

where $\text{exists}(\sigma^{+1})$ is true if σ has a successor.

Constructing Acceptors (2)

Define

$$\begin{aligned} \phi_1 \cap \phi_2 &= \wedge(\phi_1 \times \phi_2) \\ \phi_1 \cup \phi_2 &= \vee(\phi_1 \times \phi_2) \\ \phi_1 \backslash \phi_2 &= \wedge(\phi_1 \times \neg\phi_2) \end{aligned}$$

where $\wedge$ and $\vee$ are defined on pairs of booleans as expected.

Boolean border functions need to be transformed into acceptors: If ϕ is a border function with range $\mathbf{f}, \mathbf{t}$, then $\mathcal{A}[\phi]$ is defined as $(\emptyset, f_{\#}(\phi))$, where $f_{\#}(\mathbf{f}) = \#$, and $f_{\#}(\mathbf{t}) = 1$.

For example, an acceptor that accepts exactly the letters can be defined as

$$\mathcal{A}[\ (\phi_{\geq a} \cap \phi_{\leq z}) \cup (\phi_{\geq A} \cap \phi_{\leq Z})\ ].$$

Constructing Acceptors (Concatenation)

Let $\mathcal{A} = (\Lambda_1, \phi_1), \ldots, (\Lambda_n, \phi_n)$ and $\mathcal{A}' = (\Lambda'_1, \phi'_1), \ldots, (\Lambda'_n, \phi'_n)$ be acceptors.

The concatenation $\mathcal{A} \circ \mathcal{A}'$ is defined as

$$(\Lambda_1, \phi_1), \ldots, (\Lambda_n, \phi_n),\ (\Lambda'_1, \phi'_1), \ldots, (\Lambda'_{n'}, \phi'_{n'}).$$

Constructing Acceptors (Union)

Let $\mathcal{A} = (\Lambda_1, \phi_1), \ldots, (\Lambda_n, \phi_n)$ and $\mathcal{A}' = (\Lambda'_1, \phi'_1), \ldots, (\Lambda'_n, \phi'_{n'})$ be acceptors.

The union $\mathcal{A} \mid \mathcal{A}'$ is defined as

$$(\Lambda_1 \cup \{n+1\}, \phi_1), \ldots, (\Lambda_n, \phi_n),\ (\{n'+1\}, \phi_\#),\ (\Lambda'_1, \phi'_1), \ldots, (\Lambda'_{n'}, \phi'_{n'}).$$

Here, $\phi_\# = \{\, (c_\perp, \#) \,\}$.

Constructing Acceptors (Kleene Star)

Let $\mathcal{A} = (\Lambda_1, \phi_1), \ldots, (\Lambda_n, \phi_n)$ be an acceptor.

The Kleene star $\mathcal{A}^*$ of $\mathcal{A}$ is defined as

$$(\{\, 1,\, n+2 \,\}, \phi_\#),\ (\Lambda_1, \phi_1), \ldots, (\Lambda_n, \phi_n),\ (\{\, -1-n \,\}, \phi_\#).$$

Constructing Acceptors (Kleene Plus)

Let $\mathcal{A} = (\Lambda_1, \phi_1), \ldots, (\Lambda_n, \phi_n)$ be an acceptor.

The Kleene plus $\mathcal{A}^+$ of $\mathcal{A}$ is defined as

$$(\{\, 1 \,\}, \phi_{\#}),\ (\Lambda_1, \phi_1), \ldots, (\Lambda_n, \phi_n),\ (\{\, -1 - n,\ 1 \,\}, \phi_{\#}).$$

Classifiers

Assume a set T of token classes. A classifier is defined as a finite sequence of form

$$(\Lambda_1, \phi_1, t_1), \ldots, (\Lambda_n, \phi_n, t_n),$$

where each $t_i \in T$.

Obtaining Classifiers

Fix the error class $e \in T$.

Start with $\mathcal{C} = (\{\ \}, \phi_{\#}, e)$.

For every pair $(\mathcal{A}', t')$ that needs to be added, do the following:

- Let m be the current number of states in $\mathcal{C}$. Replace Λ_1 by $\Lambda_1 \cup \{m\}$.
- Remember that $e = t_1$.
- Write $\mathcal{A}'$ in the form $(\Lambda'_1, \phi'_1), \ldots, (\Lambda'_{n'}, \phi'_{n'})$.
- Append

  $$(\Lambda'_1, \phi'_1, e), \ldots, (\Lambda'_{n'}, \phi'_{n'}, e), (\emptyset, \phi_{\#}, t')$$

  to $\mathcal{C}$.

Conventions

We always assume that t_1 is the error state. This is reasonable because ϵ should be classified as error.

It must be the case that $(1, \epsilon) \vdash^* (i, \epsilon)$ implies that $t_i = t_1$. (There is no computation that reads the empty word, and which leads to a state whose classification is not t_1.)

Classification

Find the longest prefix w' of w for which there exists a computation that leads to a state i with $t_i \neq t_1$.

If no such prefix exists, classify as $(\emptyset, t_1)$.

If such w' exists, take the maximal reachable state i' which has $t_{i'} \neq t_1$, and classify as $(w', t_{i'})$.

Note that there is no order on T. That didn't work well.

nr	Λ	ϕ	T
$0:$	$\{1,4\}$	$\{\ (\$80,\#)\ \}$	**err**
$1:$	$\{\}$	$\{\ (\$80,\#),\ (A,1),(Z^{+1},\#),\ (a,1),(z^{+1},\#)\ \}$	**err**
$2:$	$\{1\}$	$\{\ (\$80,\#),\ (0,0),(9^{+1},\#),\ (A,0),(Z^{+1},\#),$ $(_,0),(_^{+1},\#),(0,0),(9^{+1},\#)\ \}$	**err**
$3:$	$\{\}$	$\{\ (\$80,\#)\ \}$	**ident**
$4:$	$\{\}$	$\{\ (\$80,\#),(w,1),(w^{+1},\#)\ \}$	**err**
$5:$	$\{\}$	$\{\ (\$80,\#),(h,1),(h^{+1},\#)\ \}$	**err**
$6:$	$\{\}$	$\{\ (\$80,\#),(i,1),(i^{+1},\#)\ \}$	**err**
$7:$	$\{\}$	$\{\ (\$80,\#),(l,1),(l^{+1},\#)\ \}$	**err**
$8:$	$\{\}$	$\{\ (\$80,\#),(e,1),(e^{+1},\#)\ \}$	**err**
$9:$	$\{\}$	$\{\ (\$80,\#)\ \}$	**while**

Determinization

Determinization is mostly free of surprises, but it is elegant.

It is easier with border functions than with intervals.

Let $\mathcal{C}$ be a classifier, let S be a set of states of $\mathcal{C}$.

- We define CLOS(S) as the smallest set of states S', s.t. $S \subseteq S'$, and $i \in S', \ j \in \Lambda_i \Rightarrow (i + j) \in S'$.
- We define
  $\text{BORD}_{\mathcal{C}}(S) = \{\sigma \in \Sigma \mid \sigma$ is in the domain of a ϕ_i with $i \in S\}$.

Conclusions

- We gave a simple representation for finite automata. Proofs are a bit more more complicated than standard, but need no further adaptation, and the implementation is simpler.
- State minimization is completely usual.

  B.t.w. the number of states removed, was a suprise for me.
- We have implemented all of this in C^{++}. See the CppNow presentation or `www.compiler-tools.eu`.

  Deterministic classifiers can be generated in table format, or as C^{++} code.

Thanks!

Thanks to Danel Batyrbek, Aleksandra Kireeva, Tatyana Korotkova, Akhmetzhan Kussainov, Olzhas Zhangeldinov.