Tokenizers

Tasks of the Tokenizer

- Group the input (which is just a stream/string of characters) into tokens. (Numbers, operators, special words, strings)
- Eliminate comments.
- In C : Deal with `#include, #if #else #endif` and the like.

Tokenizers are also called scanners.

Tokens

Definition: A token type is a pair (Λ, T), in which Λ is a finite set of tokens labels, and T is a function s.t. for each $\lambda \in \Lambda$, $T(\lambda)$ is a set.

A token (with attribute) is a pair (λ, x), s.t. $\lambda \in \Lambda$ and $x \in T(\lambda)$.

Example: If one puts $\Lambda = \{\mathbf{int}, \mathbf{real}\}$, with
$T(\mathbf{int}) = \mathcal{Z}$, $T(\mathbf{real}) = \mathcal{R}$, then
$(\mathbf{real}, 3)$, $(\mathbf{real}, 3.141526535)$, $(\mathbf{int}, 2)$, $(\mathbf{int}, -1)$ are tokens.

$(\mathbf{int}, 2.718271828)$ is not a token.

Tokens without Attribute

Not all tokens have an attribute. For example reserved words **while**, **do**, **if**, **then**, **else** usually don't.

For those, one needs a trivial type $\top = \{(\ )\}$. Then
$T(\textbf{while}) = T(\textbf{do}) = T(\textbf{if}) = T(\textbf{then}) = T(\textbf{then}) = \top$.

Implementation Issues

I usually use C^{++}. A token is a struct containing an enum type, and a list for each possible type of attribute. The list has length 1 when the attribute has the type of the list, and 0 otherwise.

In C, one could use a struct containing an enum and a pointer to the heap, or an enum with a union type.

In Java, one could use a struct containing an enum and an Object.

Whatever implemention you choose, you should use an object-oriented approach. Make sure that there is a token class, make sure that it can be printed, that it can be passed to procedures, and put in containers.

Implementation (2)

It is a good idea to add information about where it comes from to a token. This makes it more easy to generate error messages.

A tokenizer is a function with signature
`token readtoken( reader& );` When called, it reads characters from reader until it has enough information to build a token.

The reader has a field `char nextchar;` and a method
`void moveforward( );` which replaces nextchar by the next char.

Building a Tokenizer

There are basically two ways of building a tokenizer.

- Hacking (sometimes called 'careful coding by hand.') If the tokenizer is not big, you can follow this approach.
- Using a scanner generator. (Lex)

Writing a Tokenizer by Hand

If the tokens are not too many, you can follow this approach.

Draw an NDFA for each non-trivial token.

Stare at the NDFAs and the tokens for which you didn't draw an NDFA and find all overlaps.

Find ways of dealing with the overlaps. (Combine NDFAs with overlaps into one. First read one token, if NDFA gets stuck, read as another token. Do postprocessing of read tokens)

Overlaps

Sometimes, different tokens have shared prefixes.

An example is **int** and **real**. One can decide only at the end that 123453343433433434 is an **int**, and not a **real**.

Similarly, identifiers and reserved words overlap, like **while**, **do**, **dummy**, **which**.

Operators `+, ++` and `-, ->, --` overlap.

`-, --` overlap with **integer** `-1`

If you write a tokenizer by hand, you have to worry about overlaps. (which means that you loose modularity)

Usage of a scanner generator

The tokens are defined by regular expressions. The scanner generator constructs an NDFA, and translates this into an equivalent DFA. The resulting DFA is very efficient (optimal). The DFA reads the input only once. When defining the tokens, the user doesn't need to worry about overlaps.

Disadvantages are that the user has to spend time learning to use the tool, that the resulting scanner does not give much help when computing the attribute, (DFAs are only good at saying 'yes' or 'no') and that the resulting scanners are not flexible.

Non-flexibility

In general, tokenizers tend to be not as clean as parsers, and sometimes one has to use tricks.

For example in Prolog, it is important whether there is a space between an identifier and a '('.

In some languages, a `.` terminates the input, but inside `( )`, or `[ ]`, it is just a usual operator.

Non-Deterministic Finite Automata

Definition: An NDFA is a structure of form $(\Sigma, Q, Q_s, Q_a, \delta)$, in which

- Σ is the alphabet,
- Q is the set of states (finite),
- $Q_s \subseteq Q$ is the set of starting states,
- $Q_a \subseteq Q$ is the set of accepting states,
- $\delta \subseteq Q \times \Sigma^* \times Q$ is the transition relation.

We have been drawing NDFAs in the lecture and in the exercises. In a drawing, states are anonymous. If you want to represent an NDFA in a computer, you need some set Q.

NDFAs (2)

An NDFA accepts a word w iff there exist a finite sequence of words $w_1, \ldots, w_n$, and a sequence of states $q_1, q_2, \ldots, q_{n+1}$, s.t.

- $w = w_1 \cdot \ldots \cdot w_n$,
- $q_1 \in Q_s, \quad Q_{n+1} \in Q_a$,
- Each $(q_i, w_i, q_{i+1}) \in \delta$.

Non-Determinism

It would be nice if one could use use a program of form

```
state = Qs;
nextstate = delta( state, r. lookahead );
while( nextstate != undefined )
{
   r. moveforward( );  // Reads new r. lookahead
   state = nextstate;
   nextstate = delta( state, r. lookahead );
}

// Determine the type of token, based on
// the state in which we got stuck.
```

Non-Determinism (2)

Unfortunately, this is not possible, because (1) δ is not a function, but a relation, and (2) Q_s is not a single state, but a set of states.

In practice, (1) is never a problem, but (2) usually is. Problem (2) is caused by the fact that in the beginning one does not know what token will come, so one has to start with the initial states for all of them.

Remarks

NDFAs can be programmed by hand using gotos, or by keeping an explicit state variable or a set of state variables.

Tokenizers are usually greedy. This means that they try to read the longest possible token. Doing something else would be problematic.

Regular Expressions (1)

('Regular' means 'according to rules', which is actually a quite empty term.)

Let Σ be an alphabet.

- Every word $s \in \Sigma^*$ is a regular expression.
- If e is a regular expression then e^* is also a regular expression.
- If e_1, e_2 are regular expressions then $e_1 \cdot e_2$ is a regular expression.
- If e_1, e_2 are regular expressions then $e_1 \mid e_2$ is a regular expression.

Regular Expressions (2)

Other constructs can be added as well:

- If e is a regular expressions, $n \geq 0$, then e^n is a regular expression.
- If e is a regular expression, then $e?$ is a regular expression.
- If e is a regular expression, then e^+ is a regular expression.
- If the alphabet Σ is ordered by a total order $<$, and $\sigma_1, \sigma_2 \in \Sigma$ $\sigma_1 \leq \sigma_2$, then $\sigma_1 \ldots \sigma_2$ is a regular expression.

All these constructions are definable, but the definitions can be quite long. For example, $e^{20} = e \cdot \ldots \cdot e$.

$\sigma_1 \ldots \sigma_2 = \sigma_1 \mid \sigma_1' \mid \sigma_1'' \mid \sigma_1''' \mid \cdots \mid \sigma_2$. (This is only possible if Σ is finite)

Regular Expressions (3)

Examples:

```
digit := '0' .. '9'
letter := 'a' .. 'z' | 'A' .. 'Z'
ident := letter ( letter | digit | '_' ) +
float := ( "" | "+" | "-" )
         digit +
         ( '.' digit + ) ?
         (( 'e' | 'E' ) ( '-' | '+' | "" ) digit + ) ?
```

What do you find more readable? NDFAs or regular expressions?

Regular Expressions (4)

We define recursively when a word in Σ^* satisfies a regular expression:

- If s is a regular expression built by a single word w and $w = s$, then w satisfies s.
- If w is a word, then w satisfies e^* if either $w = \epsilon$, or there exist w_1, w_2, such that $w = w_1 \cdot w_2$, w_1 satisfies e and w_2 satisfies e^*.
- If w is a word, then w satisfies $e_1 \cdot e_2$ if there exist w_1, w_2, s.t. $w = w_1 \cdot w_2$, w_1 satisfies e_1, and w_2 satisfies e_2.
- If w is a word, then w satisfies $e_1 \mid e_2$ if either w satisfies e_1 or w satisfies e_2.

This is not a logic course, but the logically correct definition would be: satisfies(w,e) := the $\subseteq$-smallest 2-place binary predicate that has the list of properties mentioned above.

Structure of a Scanner Generator

A scanner generator proceeds as follows:

1. Translate the regular expressions belonging to the tokens into NDFAs.
2. Combine the NDFAs for the tokens into a single NDFA.
3. Translate the NDFA into a DFA.
4. Minimize the DFA.
5. Compress the DFA and generate tables.

Translating a Regular Expression into an NDFA

Translating regular expressions into NDFAs is surprisingly easy.

For a regular expression e, the translation $\mathcal{A}(e) = (\Sigma, Q, Q_s, Q_a, \delta)$ will be defined on the following slides.

Σ will be always the same.

$\mathcal{A}$ is defined by recursion on the structure of e.

Translating a Regular Expression into an NDFA

Assume that e is built from a single word s. The translation $\mathcal{A}(e)$ is the automaton $(\Sigma, \{q_s, q_a\}, \{q_s\}, \{q_a\}, \{(q_s, s, q_a)\})$.

Translating a Regular Expression into an NDFA

Asume that e has form $e = e_1 \cdot e_2$. Let

$$A_1 = \mathcal{A}(a_1) = (\Sigma, Q_1, Q_{1,s}, Q_{1,a}, \delta_1),$$

and let

$$A_2 = \mathcal{A}(a_2) = (\Sigma, Q_2, Q_{2,s}, Q_{2,a}, \delta_2).$$

Assume that Q_1 and Q_2 have no states in common. Otherwise, rename the states in A_1.

$\mathcal{A}(e_1 \cdot e_2) = (\Sigma, Q, Q_s, Q_a, \delta)$ is obtained as follows:

- $Q = Q_1 \cup Q_2$,
- $Q_s = Q_{1,s}, \quad Q_a = Q_{2,a}$,
- $\delta = \delta_1 \cup \delta_2 \cup \{(q, \epsilon, q') \mid q \in Q_{1,a},\ q' \in Q_{2,s}\}$.

Translating a Regular Expression into an NDFA

For a regular expression e of form $e = e_1 \mid e_2$, let

$$A_1 = \mathcal{A}(e_1) = (\Sigma, Q_1, Q_{1,s}, Q_{1,a}, \delta_1),$$

and let

$$A_2 = \mathcal{A}(e_2) = (\Sigma, Q_2, Q_{2,s}, Q_{2,a}, \delta_2).$$

Assume that A_1 and A_2 have no states in common. If they have, then rename the states in A_1. Then $\mathcal{A}(e_1 \mid e_2) = (\Sigma, Q, Q_s, Q_a, \delta)$ is obtained as follows:

- $Q = Q_1 \cup Q_2$,
- $Q_s = Q_{1,s} \cup Q_{2,s}, \quad Q_a = Q_{1,a} \cup Q_{2,a}$.
- $\delta = \delta_1 \cup \delta_2$.

Translating a Regular Expression into an NDFA

For a regular expression e of form $e = e_1^*$, let

$$A_1 = \mathcal{A}(e_1) = (\Sigma, Q_1, Q_{1,s}, Q_{1,a}, \delta_1).$$

Then $\mathcal{A}(e_1^*) = (\Sigma, Q, Q_s, Q_a, \delta)$ is obtained as follows:

- $Q = Q_1 \cup \{\hat{q}\}$
- $Q_s = \{\hat{q}\}, \quad Q_a = \{\hat{q}\},$
- $\delta = \delta_1 \cup \{(\hat{q}, \epsilon, q) \mid q \in Q_{1,s}\} \cup \{(q, \epsilon, \hat{q}) \mid q \in Q_{1,a}\}.$

Translating a Regular Expression into an NDFA

Theorem Let e be regular expression. Then w satisfies e iff $\mathcal{A}(e)$ accepts w.

Deterministic Finite Automata

Definition: An NDFA $\mathcal{A} = (\Sigma, Q, Q_s, Q_a, \delta)$ is called deterministic if

1. Q_s contains at most one element,
2. $(q, s, q') \in \delta \Rightarrow s$ has length 1.
3. $(q, s, q_1), (q, s, q_2) \in \delta \Rightarrow q_1 = q_2$.

In summary, a DFA always knows which transition to make when it sees the next token.

Determinization

In the slides that follow, we present a procedure that transforms an NDFA into an equivalent DFA.

Simplification of δ

In our definition of NDFA, it is allowed to have transitions of form (q, w, q') in δ, where $|w| \geq 2$.

The first step is to eliminate these transitions. Let $\mathcal{A} = (\Sigma, Q, Q_s, Q_a, \delta)$ be an NDFA.

- As long as δ contains a transition (q, w, q') with $|w| \geq 2$, do the following: Write $n = |w|$. Let $q_1, \ldots, q_{n-1}$ a sequence of new states, not in Q. Put

$$Q := Q \cup \{q_1, \ldots, q_{n-1}\},$$

and put

$$\delta := \delta \backslash \{(q, w, q')\} \cup \{(q, w_1, q_1), (q_1, w_2, q_2), \ldots, (q_{n-1}, w_n, q')\}.$$

Outline (1)

If you have a non-deterministic automaton $\mathcal{A} = (\Sigma, Q, Q_s, Q_a, \delta)$, then for every word $w \in \Sigma^*$, there exists a set of reachable states $Q' \subseteq Q$, which is obtained as follows:

A state q is reachable under w if there exists a finite sequence of words $w_1, \ldots, w_n$, and a sequence of states $q_1, q_2, \ldots, q_{n+1}$, s.t.

- $w = w_1 \cdot \ldots \cdot w_n$,
- $q_1 \in Q_s, \quad q_{n+1} = q$,
- Each $(q_i, w_i, q_{i+1}) \in \delta$.

(Intuitively, the state q is reachable under w if the automaton can start in a starting state, eat the word w, and end up in state q)

Outline (2)

The algorithm explores all sets of reachable states $R \subseteq Q$ and constructs the graph of them.

Since there are only finitely many subsets of Q, this exploration will eventually end, and the resulting graph will be a deterministic finite automaton.

Epsilon Closure

Let $S \subseteq Q$ a set of states belonging to an NDFA $\mathcal{A} = (\Sigma, Q, Q_s, Q_a, \delta)$. The ϵ-closure of S is the smallest set S', s.t.

- $S \subseteq S'$,
- If $q \in S'$ and $(q, \epsilon, q') \in \delta$, then $q' \in S'$.

CLOS(S) can be computed as follows:

- $S' := S$,
- As long as there exist $q \in S'$ and $(q, \epsilon, q') \in \delta$, s.t. $q' \notin S'$ do $S' := S' \cup \{q'\}$,
- Now $S' = \text{CLOS}(S)$.

Step Function

Let $S \subseteq Q$ be a set of states belonging to an NDFA $\mathcal{A} = (\Sigma, Q, Q_s, Q_a, \delta)$. Let $\sigma \in \Sigma$. Then STEP(S, σ) is defined as the set

$$\{q' \mid \text{ there is a } q \in S, \text{ s.t. } (q, \sigma, q') \in \delta\}.$$

Let $\mathcal{A} = (\Sigma, Q, Q_s, Q_a, \delta)$ be an NDFA. The determinization of $\mathcal{A}$ is the automaton $\mathcal{A}' = (\Sigma, Q', Q'_s, Q'_a, \delta')$, which is the result of the following algorithm:

- Start with $\mathcal{A}' := (\Sigma,\ \{\mathrm{CLOS}(Q_s)\},\ \mathrm{CLOS}(Q_s), \emptyset, \emptyset)$.
- As long as there exist an $S \in Q'$, and a $\sigma \in \Sigma$, s.t. $S' = \mathrm{CLOS}(\mathrm{STEP}(S, \sigma)) \notin Q'$, put

$$Q' := Q' \cup \{S'\}, \quad \delta' := \delta' \cup \{(S, \sigma, S')\}.$$

  If $Q_a \cap S' \neq \emptyset$, then also

$$Q'_a := Q'_a \cup \{S'\}.$$

- As long as there exist $S, S' \in Q'$, and a $\sigma \in \Sigma$, such that $S' = \mathrm{CLOS}(\mathrm{STEP}(S, \sigma))$ and $(S, \sigma, S') \notin \delta$, put

$$\delta := \delta \cup \{(S, \sigma, S')\}.$$

Minimalization of a DFA

It can happen that the DFA that was obtained by the previous construction, is not minimal. Such a DFA will appear if one determinizes the NDFA resulting from the following regular expression: $(ab|(ab)^*)^*$.

On the following slides we will give a procedure for detecting states with the same observational behaviour. Once the states are found, they can be unified, which results in an automaton with less states.

Definition: Let $(\Sigma, Q, Q_s, Q_a, \delta)$ be a DFA. A state partition Π is a set of sets of states with the following properties:

- For every q in Q, there is an $S \in \Pi$, s.t. $q \in S$.
- For every $q \in Q$, if there are $S_1, S_2 \in \Pi$, s.t. $q \in S_1$ and $q \in S_2$, then $S_1 = S_2$.

So Π separates Q into different groups. Each $q \in Q$ occurs in exacty one group.

The aim is to construct Π in such a way that states that 'behave in the same way' go into the same group.

Initially, all states are put in a single group. Then all groups are inspected for states that behave different in some way. If such states are found, the group is separated into two new groups. The procedure stops when no more separations are possible.

Two states have different behaviour if

1. One of them is an accepting state, while the other is not
2. There is a letter $s \in \Sigma$, such that the transitions from the states end up in states that are in different partitions.
3. There is a letter $s \in \Sigma$, such that from one of the states a transition is possible, while from the other it is not.

Minimalization Algorithm (Initial Partition)

- The algorithm starts with the partition

$$\Pi := \{Q \backslash Q_a, Q_a\}.$$

If different elements in Q_a accept different tokens, one has to further partition Q_a according to the tokens that are being accepted.

For example if Q_a consists of three states q_1, q_2, q_3, where q_1 accepts **real**, and q_2, q_3 accept **int**, then one has to start with the partition

$$\Pi := \{\ Q \backslash \{q_1, q_2, q_3\}, \{q_1\}, \{q_2, q_3\}\ \}.$$

Minimalization Algorithm (Refining the Partition)

- As long as there exist $S, S' \in \Pi$, states $q_1, q_2 \in S$, and a $\sigma \in \Sigma$, and a state $q_1' \in S'$, s.t.

$$(q_1, \sigma, q_1') \in \delta,$$

while at the same time there is no state $q_2' \in S'$, s.t.

$$(q_2', \sigma, q_2') \in \delta,$$

replace S in Π by two sets as follows:

$$\{q \in S \mid \text{ there is a } q' \in S', \text{ s.t. } (q, \sigma, q') \in \delta\},$$

and

$$\{q \in S \mid \text{ there is no } q' \in S', \text{ s.t. } (q, \sigma, q') \in \delta\}.$$

Minimalization Algorithm (Reading Of the Result)

Let $\mathcal{A} = (\Sigma, Q, Q_s, Q_a, \delta)$ be a DFA. Let Π be the partition constructed by the determinization algorithm. Then the simplified automaton $\mathcal{A}' = (\Sigma, Q', Q'_s, Q'_a, \delta')$ can be constructed as follows:

- $Q' = \Pi$,
- $Q'_s = \{S \in \Pi \mid Q_s \in S\}$,
- $Q'_a = \{S \in \Pi \mid Q_a \in S\}$,
- $\delta' = \{(S, s, S') \mid$ there are $q, q' \in S, S',$ s.t. $(q, s, q') \in \delta\}$.

Pruning the DFA

Let $(\Sigma, Q, Q_s, Q_a, \delta)$ a (N)DFA. If Q contains states that

1. are not reachable from Q_s, or
2. from which there exists no path to a state in Q_a,

then remove these states, and all the transitions in δ in which these states occur.

FLEX tool

The FLEX tool reads a list of regular expressions and associated actions.

It performs the minimal DFA construction listed on the previous pages.

Usage of FLEX tool (my impression)

- It is really very easy to write a complex scanner with FLEX.
- I find the syntax of FLEX not so good.
- FLEX gives no support in the computation of the attributes. One still has to use **atoi**, **atof**. This means that one still uses an NDFA that somebody wrote by hand. (But you have seen in the exercises that the main problem is in the combination of different tokens. At least this problem was solved by FLEX)
- The C^{++} interface is not so good? C^{++} is more than a few plusses on C. It is another programming paradigm, and I don't see this in FLEX.