Tokenizing

Copyright and AI Notice

These slides are intended for studying and teaching at Nazarbayev University in Astana, Kazakhstan.

If you want to redistribute or adapt these slides for different purposes, check the license first.

No generative AI was used in the preparation of these slides.

Tasks of the Tokenizer

- Group the input (which is just a stream/string of characters) into tokens. (Numbers, operators, special words, strings)
- Eliminate comments.

Tokenizers are also called scanners or lexers.

Tokens

Definition: A token is a pair (λ, a), where λ is the label of the token, and a is the attribute of the token. We write Λ for the set of possible labels, which is a finite set. The type of a depends on the value of λ. We write $T(\lambda)$ for the type that a must have, when the label equals λ.

Example: Let $\Lambda = \{\, \textbf{int}, \textbf{real} \,\}$, $T(\textbf{int}) = \mathcal{Z}$, $T(\textbf{real}) = \mathcal{R}$. Then $(\textbf{real}, 3.0)$, $(\textbf{real}, 3.1415926535)$, $(\textbf{int}, 2)$, $(\textbf{int}, -1)$ are tokens.

$(\textbf{int}, 2.718281828)$ is not a token, because 2.7182818 is not of type $\mathcal{Z}$.

Tokens without Attribute

Not all tokens need an attribute. For example reserved words (like **while**, **do**, **if**, **then**, **else**)

For those, one needs a trivial type **unit**. So
$T(\textbf{while}) = T(\textbf{do}) = T(\textbf{if}) = T(\textbf{then}) = T(\textbf{else}) = \textbf{unit}$.

unit is type that consists of exactly one element, which I will call **trivial**.

Since **trivial** carries no information, we will write λ instead of $(\lambda, \textbf{trivial})$.

Implementation Issues

Avoid using *C*. Use a language that has support for dependent type union. The following languages have this:

- Java, $C^{\#}$: Use subclasses of a base **attribute** type.
- C^{++} : Use `std::variant`. You could use subclasses but don't.
- Rust: Has a built-in **enum** type which directly supports dependent union.

Make sure that tokens can be printed, and put in containers.

Implementation (2)

It is a good idea to add source information to a token. This makes it easier to generate error messages. This information should include file name, position in the file, and possibly the include path.

Dealing with includes can be pretty unpleasant. You will need to think about how to store a source location.

Building a Tokenizer

There are two ways of building a tokenizer:

- Program the whole thing by hand. If the tokenizer is not big and the language is unlikely to change, you can do this.
- Use a tokenizer generator, for example (Maphoon).

In many cases, a programmer will start first approach, be he/she does not like to study documentation (theory), and regret later.

Writing a Tokenizer by Hand

This may seem the best approach, when there are not many tokens.

Draw an NDFA for the complicated tokens. These are usually quoted strings, numbers, and comments.

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

Find a way of dealing with the shared prefixes (Combine NDFAs with shared prefix into one. First read one token, if NDFA gets stuck, read as another token. Do postprocessing of read tokens. Be careful not to become exponential, by careless search.)

Non-Deterministic Finite Automata

If you write a tokenizer by hand, you will need NDFAs for some tokens.

You have seen NDFAs during CSCI 272.

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.

NDFA (2)

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

It is usually easy to write an implementable NDFA for a single token, but hard to write a recognizer for all tokens combined.

This is because of shared prefixes.

Shared Prefixes

Different tokens have shared prefixes.

An example is **int** and **real**. One can decide only at the end that 123453343433433434.5 is a **real**, and not an **int**.

Similarly, identifiers and reserved words overlap, like **while**, **do**, **double**, **int**.

Operators `+, ++` and `-, ->, --` have shared prefixes.

`-, --` shares a prefix with **integer** `-1`

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

Combining Tokens

We assume that we are able to write recognizers for each of the tokens separately. In order to obtain a complete tokenizer, one can try to use the following tricks:

- Look at the next character and decide which recognizing function to call. In this way, one can easily separate between identifiers and numbers.
- Design recognizing functions in such a way that they can fail gracefully. (e.g. by recognizing a token of length 0.)

  For example, one can first try to recognize an integer, and if that fails, try recognize a double.
- Postprocess recognized tokens. For example, one can first read an ordinary identifier, and afterwards check if should be changed into a reserved word.

Usage of a Tokenizer Generator

With a tokenizer generator, the tokenizer can be defined by means of regular expressions. The user provides a list of regular expressions together with the token that they define.

The tokenizer generator creates an NDFA for each of the expressions, and merges them into a single 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 shared prefixes, because the DFA-construction deals with this.

Disadvantages are that the user has to spend some time learning to use the tool, and the user still has to provide code that computes attributes.

More than Automata

Many programming languages have requirements on the tokenizer that cannot be expressed with automata:

In Prolog, it matters if there is a space between an identifier and a '('.

In Python, indentation matters, so one has to create a token when the indentation increases, or decreases. If indentation decreases it must be down to a level that was used before.

In Prolog, a dot (`.`) terminates the input, but inside `( )` or `[ ]`, it is an operator.

In C^{++}-11, a `>>` can denote the `>>`-operator, or two separate occurrences of `>`, as in `std::list< std::list< int >>`.

Regular Expressions (1)

('Regular' means 'according to rules', which is actually a quite meaningless word.)

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. (Usually, one just writes $e_1\, e_2$.)
- If e_1, e_2 are regular expressions then $e_1 \mid e_2$ is a regular expression.
- $\bot$ is a regular expression. It accepts nothing.
- $\top$ is a regular expression. It accepts every $\sigma \in \Sigma$.

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 $\sigma_1 \cdots \sigma_2$ is a regular expression.

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

$$\sigma_i \cdots \sigma_k = \sigma_i \mid \sigma_{i+1} \mid \sigma_{i+2} \mid \cdots \mid \sigma_{k-1} \mid \sigma_k.$$

(This is only possible if Σ is finite, and not too big.)

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?

Structure of a Tokenizer Generator

A tokenizer generator works as follows. The user prepares a table or regular expressions, with token names attached to them. The token names must have priorities, because sometimes tokens can be overlapping (e.g. reserved words and ordinary identifiers.)

1. Translate the regular expressions into NDFAs. Mark each accepting state with its token name.
2. Combine the NDFAs for the tokens into a single NDFA.
3. Translate the NDFA into a DFA. In case different accepting states are merged into a single state, the new state will accept the token with the highest priority.
4. Minimize the DFA.
5. Generate tables or executable code.

Tokenizer Generator: Resulting Tokenizer

The tokenizer runs the automaton until it gets stuck.

At this point, it looks back to the last time it was in an accepting state.

This determines the token recognized, and how long it is.

Maphoon

We will be using a tool for token generation that I constructed.

For the reasons explained on Slide 14, it does not construct a complete tokenizer.

It constructs something called classifier, which reads input, tells the type of token recognized, and its length.

Filereader

The interface to Maphoon is through a filereader (which is not a type but a concept):

- `bool has( size_t len );` Try to make sure that the buffer has at least `len` characters, and return `true` if this succeeded.
- `char peek( size_t i ) const;` See the i-th character in the buffer. Size of the buffer must be $> i$.
- `void commit( size_t i );` Remove the first `i` characters from the buffer. Size of the buffer must be $\geq i$.
- `std::string_view view( size_t i ) const;` Get a view to the first i characters. Size of the buffer must be $\geq i$.

String View

A `std::string_view` contains a pointer to `char` and a `size_t`.

It is parasitic object that uses the characters stored somewhere else. In the case of `filereader::view` these characters are in the buffer of `filereader`.

It is a very dangerous datastructure, because the stored characters may disappear, and the `std::string_view` wouldn't notice.

For the `std::string_view` returned by `view`, don't call `commit` or `has` as long as the `std::string_view` exists.

Immediately convert the `std::string_view` into something else. If you want to keep the string, convert it to `std::string`.

Using Filereader

Define a tokenizer class, let `lexing::filereader` be a field of this class.

Take some care with the constructor, because `filereader` cannot be copied. It can be moved only.

Decide what type you want to use for classification. This is normally an **enum** type, but you can also use **int** or `std::string`. (I use string during the lecture, because efficiency does not matter and strings are easy to handle.)

Write functions of form `try_X( )`, that return a `std::pair< symbolval, size_t >`, containing the classification together with its size.

Failure can be expressed by returning an error classification, or by returning size = 0.

Maphoon for Lexing

First decide on the alphabet (nearly always `char`).

Maphoon will create a function `readandclassify` that follows the interface on the previous slide, using regular expressions.

Maphoon does not directly operate on regular expressions. Instead, it views regular expressions directly as automata, called acceptors.

In the end, all acceptors are combined into a classifier, which is a combination of acceptors, together with the symbol they define.

Maphoon for Lexing (2)

Create

```
auto cls = classifier< char, symboltype > cls( err );
   // The constructor needs the error token. This is
   // the token  will be returned when nothing can
   // be recognized.
   // symboltype is the type you defined, and err must be
   // of this type.

   // A classifier consists of pairs of
   // acceptors and symboltype.

   // In order to add a pair, use
   // .insert( acceptor , symboltype ).
```

Maphoon for Lexing (3)

```
range( c1, c2 );  // Acceptor that recognizes a range.
just(c);          // Acceptor that recognizes c.
word(w);          // Acceptor that accepts w.
|                 // as in regular expressions.
*                 // concatenation.
.star( )          // The star from regular expressions.
.plus( )          // plus from regular expressions
.optional( )      // ? from regular expressions.
every<char> ( )   // Every character.
.without(c);      // Remove c from acceptor.
.without(c1,c2);  // Remove range c1..c2 from acceptor.
                  //    (Not regular operations, but
                  //     sometimes useful)
empty( );         // Acceptor that accepts nothing.
epsilon( );       // Acceptor that accepts empty word.
```

Maphoon for Lexing (4)

Sometimes it is useful to define priorities between classifications. For example, reserved words (like `while`, `if`, ...) can also classify as identifiers.

Maphoon ranks classifications by order of appearance: Classifications that were introduced later obtain higher priority.

Reserved words should be defined after identifiers, and integers should be defined after floating point numbers. (Because an integer can be a floating point number).

With the classifier, call `readandclassify( cls, inp )`, where `cls` is the classifier and `inp` is the file reader.

It returns a `std::pair< symboltype, size_t >` as explained before.

For efficiency, you can make the classifiifier deterministic by calling

```
cls = make_deterministic( cls );
cls = minimize( cls );
```

Make sure that the classifier is not recomputed every time the is tokenizer is called. Create a function that constructs it, and put the classifier in a static variable, like

```
static auto cls = buildclassifier( );
```

Defining your Own Token Class

Maphoon can automatically create the token class (confusingly called `symbol`), but if you don't want that, write something like:

```
struct token {
   enum {    } type;
   std::variant< ... > attr;
      // the possible attribute types.
};
```

- Don't use OO (derived classes), use `std::variant` instead.
- Decide about copyability. Attribute types do not need to be copyable. In that case, they must be movable.
- Define suitable constructors, and `operator <<` , also for the `enum` type.
- Decide about source file information. This can be tricky in case

of include files.

Summary

The tokenizer must:

Interact with the input source. (Either a file, or the user.)

Group the input into tokens.

Construct the attributes of these tokens.

Possibly look up identifiers.

Take context into account, (unfortunately often) because not all tokens can occur in all contexts.

Attach source information to the token. (File, line number, position) This information must be preserved through the compilation process, until is certain that no further errors will occur.