Wednesday, August 25, 2021 1:34 PM

last time - Intro to syntax using the english language as motivation
- started with programming languages syntax

Today — syntax of programming languages
— getToken() function

---

alphabet ✓

strings ✓

token : A token is a set of strings.
You can think of the token as the label or the name of the set.

Example 1 NUM = { "0", "1", .., "10", "11", ... }
integer constant

Example 2 ID = the set of strings
identifier consisting of 1 or more alphabetical characters, underscore, or digits that start with an underscore or an alpha character

Examples of strings in ID

_ ✓ _1 ✓ _a ✓ ~~1_~~ ✗

✓ ✓ ~ ^

Example 3 DECIMAL : the set of strings of the form NUM "." NUM or "." NUM or NUM "."

1.1 ✓ 1. ✓ .1 ✓ .. ✗

lexeme A lexeme of a token is a string in the set of strings of the token

Example "123" is a lexeme of NUM
"abc" is a lexeme of ID
"11.12" is a lexeme of DECIMAL

Oftentimes we refer to a lexeme as a token
So we say 11.12 is a DECIMAL token

Syntax vs. Semantics

Syntax : only the tokens (categories) are relevant

Semantics : the lexemes are relevant

Example The cat drank the milk.
The milk drank the cat.

Example

The milk drank the cat.

Both sentences are syntactically identical.

DEFART NOUN VERB DEFART NOUN DOT

Semantically they are very different:

One makes sense and one is nonsense

In programming languages, we make the same distinction between syntax and semantics.

Example declaration : ID COLON ID SEMICOLON

assign_stmt : ID EQUAL ID SEMICOLON

```
x : int ;      //  ID COLON ID SEMICOLON
y : int;       //  ID COLON ID SEMICOLON
x = y ;        //  ID EQUAL ID SEMICOLON
```

Syntactically the same as

```
x : int ;      //  ID COLON ID SEMICOLON
y : boolean;   //  ID COLON ID SEMICOLON
x = y ;        //  ID EQUAL ID SEMICOLON
```

but semantically different especially for a language that does not allow assignment between int and boolean.

---

getToken() function

getToken() function

Given a list of tokens (categories), we are interested in breaking down the input into a sequence of tokens from the list and their associated lexemes.

getToken() takes input from standard input

returns a struct that has two fields:

token_type: the category

lexeme: the actual part of the input that corresponds to the identified token

lineno: the line at which token/lexeme appears in the input. I will ignore this in the examples below

Depending on the language, getToken() might ignore (skip over) some parts of the input (like blank space).

Example

token list

ID
EQUAL

EQUAL
SEMICOLON

input

x = y ;

Call 1. getToken() ⟶ (ID, "x")

Call 2. getToken() ⟶ (EQUAL, "=")

Call 3. getToken() ⟶ (ID, "y")

Call 4. getToken() ⟶ (SEMICOLON, ";")

Call 5. getToken() ⟶ (EOF, "")

↑ to indicate that there are no more characters in the input

Call 6. getToken() ⟶ (EOF, "")

Example token_list

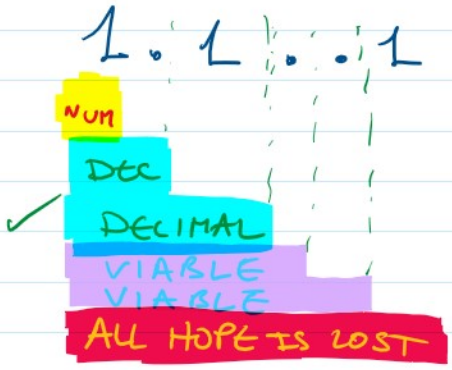
NUM
DOT
DECIMAL (in the typed notes I give an example in which I modify the definition of DECIMAL)

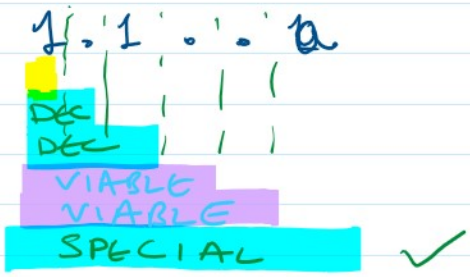
SPECIAL = {"1.1..a"}

input 1.1..1

Call 1. getToken() ⟶

a = 1.2;





we have more than one prefix of the remaining input that is a valid token

The longest possible prefix rule : getToken()
returns the token with the longest lexeme

Example. token list

IF = { "if" }

ID

input.

if 1 if if

Call 1. getToken() —> (ID, "if 1 if")
Call 2. getToken() —> (IF, "")

↑ IF appears before ID in the list of tokens

Rule If the longest possible prefix corresponds to more than one token, we return the one that appears first

corresponds to more than one token,
we return the one that appears first
in the list.