Seminar 11: Language Syntax

Language Aspects (20 minutes)

There are three aspects to every programming language: syntax, semantics, and pragmatics.

Syntax

The syntax determines which programs are considered "well-formed" in a particular language.

Examples of syntactically incorrect programs:

```
int x = "hello" ; // type mismatch

int x = y + 23 ; // invalid if y is not declared

int int = 4;   // after int, a variable name is expected, which cannot match
the int keyword
```

Semantics

Semantics determines the meaning of language constructs. It can be described in natural language (for example, in the C language standard it is described in English) or in the language of formal logic. Semantics endows each construct with some behavior depending on the context, for example, matches the operation “*” with the meaning “multiply the left side by the right side if the asterisk is between two expressions”, or the meaning “to the left of the asterisk is a pointer type” in another context.

It happens that some construction does not make sense, although the program is syntactically correct. “It does not make sense” means the language standard either explicitly refused to determine what will happen, or simply bypassed this point. This situation is called *undefined behavior*.

Examples of programs that lack semantics:

```
int y = 0;
int x = 1 / y; // division by zero

int z = * (NULL); // reference to a NULL pointer is undefined.


int k = x++ + ++x; // it is not known which operation will be performed
first: // x ++ or ++ x, hence the result may be different.
```

Pragmatics

Pragmatics in programming languages corresponds to the semantics change when translated according to the target architecture. For example, a C program has undefined behavior, but an assembly program does not. This means that after the compilation, the undefined behavior will be redefined – but we do not know how exactly. Some additional directives for the compiler and language construct are needed to take into account the peculiarities of the computer, implementation, to improve efficiency.

Examples of pragmatic aspects:

- Alignment in memory
- Packing the fields of structures (how are the fields located in memory, is there any indentation between them?)
- Choice of translation method for floating point formulas.

For example, when compiling, `gcc` can accept the `-ffast-math` option, which:

- Turns off strict compliance with the IEEE standard
- Reduces the number of entries to the `errno` variable.
- Makes the assumption that no NaN, zero, or infinity occurs in formulas.
- And includes some more optimizations.

Question 1: compile the following file and view the contents of the `.data` section with `objdump`. At what address does the variable `x` start? Try to remove _Alignas (128), do previous steps again and explain the effect. Send your explanation through the form.

```
#include <inttypes.h>

_Alignas(128) int64_t x = -1;

int main() {
  return 0;
}
```

Here _Alignas () sets the Alignment

Every complete object type has a property called *alignment requirement*, which is an integer value of type `size_t` representing the number of bytes between successive addresses at which objects of this type can be allocated. The valid alignment values are non-negative integral powers of two.

Syntax (50 minutes)

We are used to looking at programs as a text. However, the textual representation has several disadvantages.

First of all, the text can be interpreted ambiguously.

What do you think this program will output if `x = -4`?

```
if (x > 0)
  if (y > 0) {
    print("yes");
  }
  else {
    print("no");
  }
```

And if you look at it like that?

```
if (x > 0)
  if (y > 0) {
    print("yes");
  }
else {
  print("no");
}
```

Question 2 Write a minimal example with this code and compile it strictly according to the C17 standard `(-std = c17 -pedantic -Wall)`. What will the compiler output? Explain the message. Send the output of a program and your explanation via the form.

As we can see from this example, in poorly designed languages, the textual representation is ambiguous.

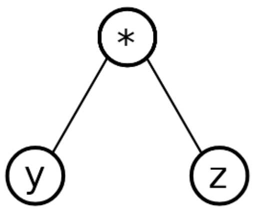
Moreover, many elements of the program do not carry any operational load (comments, keywords). Some apparently identical programs can be represented in different text:

```
int x = a * b + c;
int x = (a * b) + c;
int x = ((a * b) + c);
int x = (((a * b) + c));
// parentheses are needed only to parse the expression correctly
// but they are redundant here
// this comment does not affect anything either,
// although it is present in the program text.
```

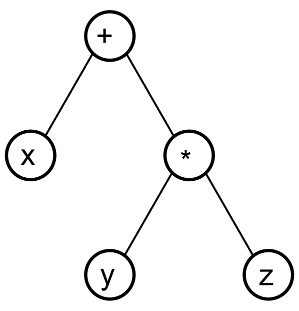
It is more convenient to think of a program not as its textual representation. The text is just a projection of its structure onto the screen; in fact, a better idea of its structure is given by the so-called *abstract syntax tree*, AST.

In it, each structure of the language is associated with the type of tree vertices, for example::

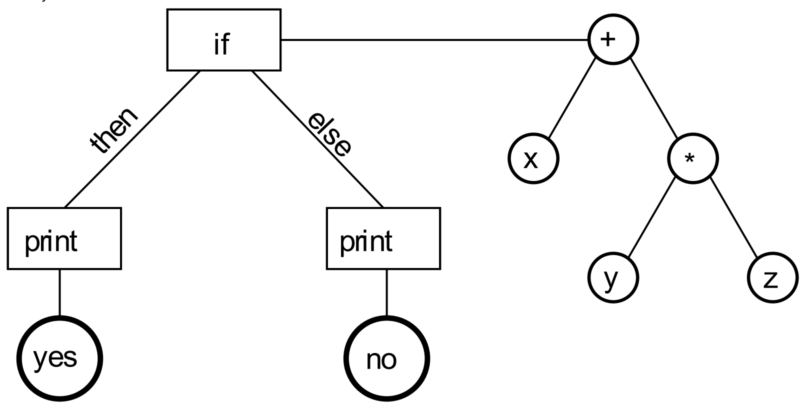
`y*z`



`x+y*z`



```
if (x+y*z) {
  print("yes");
  }
else {
  print("no");
  }
```



The meaning of such a program representation is that it clearly and unambiguously defines its structure and does not depend on formatting (spaces, line breaks, etc.). In addition, it does not depend on the exact names of the keywords. Indeed, will C become different if all we change is to replace the `while` keyword with `whiiile` with the same meaning?

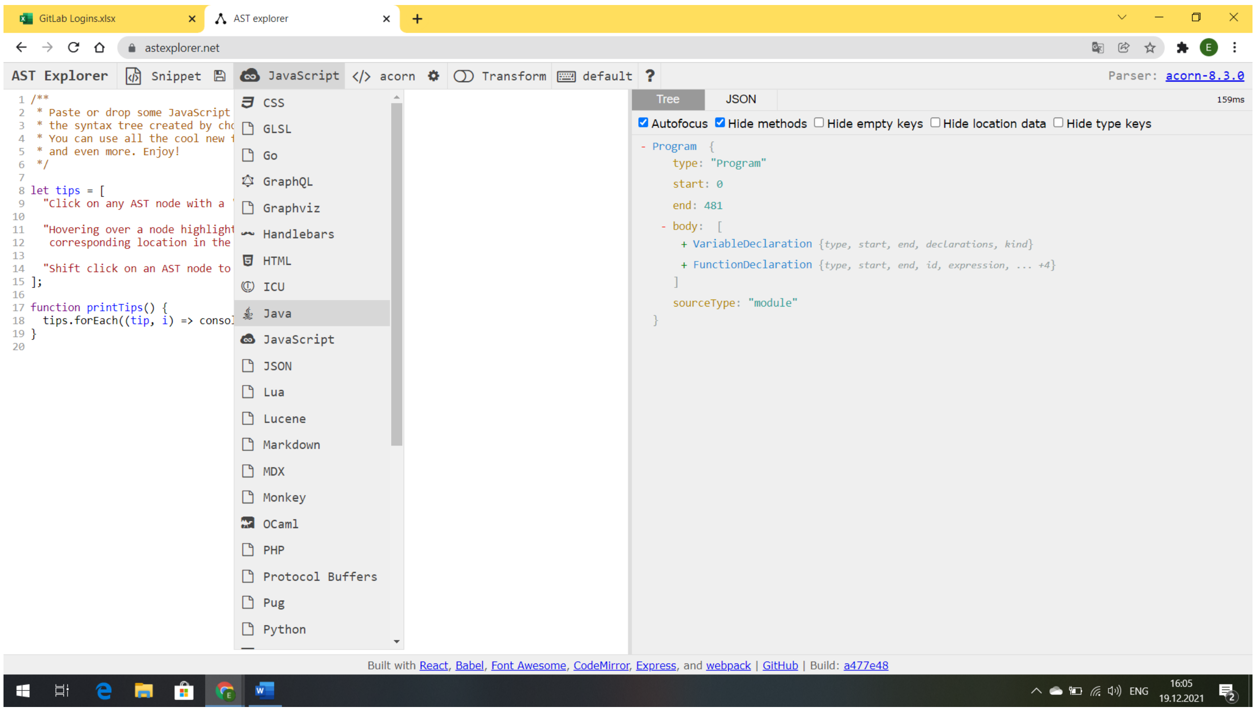
With such a tree-like representation of the program, it is easier to talk about its semantics. The meaning of the language constructs is set specifically for the vertices of the AST.

What is it used for:

- When developing compilers or interpreters. Practically the first thing that happens to a program is its conversion to AST. This is also what IDEs and text editors do.
- Data storage formats like JSON are also representable as AST.
- Text markups (Markdown, Org ...)
- In general, any structured text formats (configuration files, etc.)
- In documents describing programming languages (language standards).
- Knowledge of the AST is useful when trying to understand the error messages produced by the compiler

Now let's explore the AST of a simple Java program using https://astexplorer.net/

```
public class Program {
    public static void main(String[] args) {
        int x = 100;
        if (x > 0) {
            System.out.println("Greater than zero");
        }
        else {
            System.out.println("Less than or equal to zero");
        }
    }
}
```

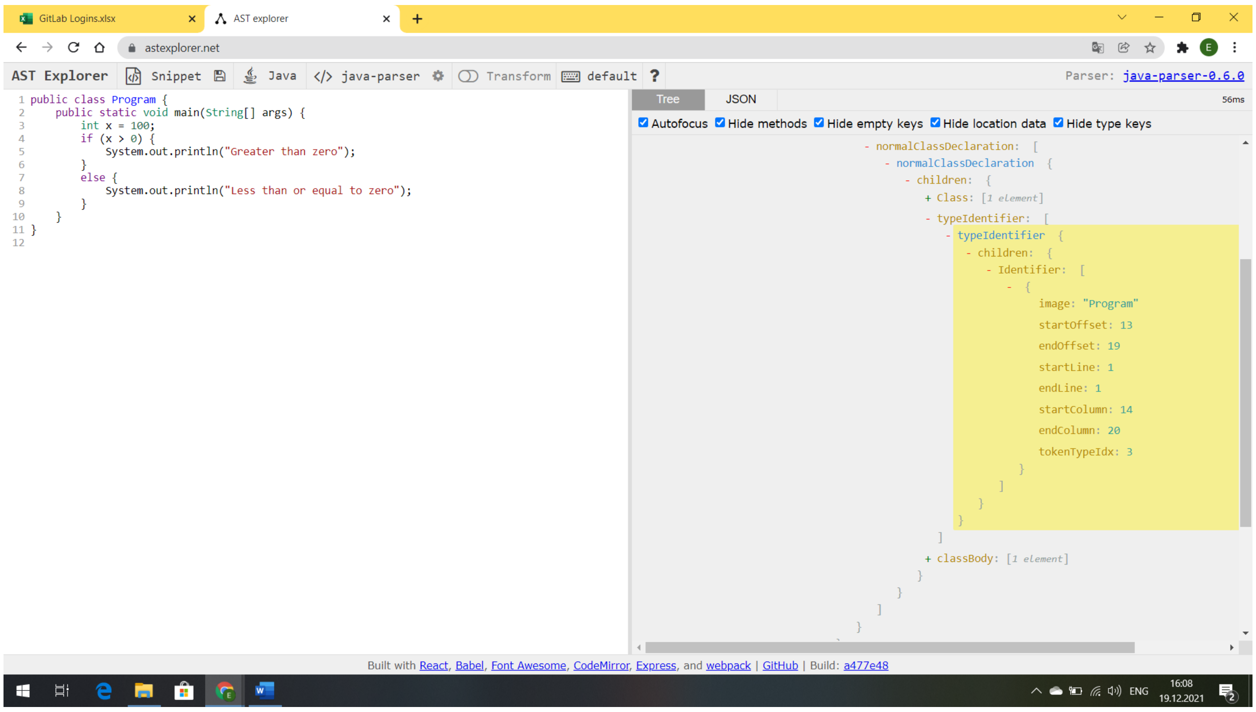


```
/**
 * Paste or drop some JavaScript
 * the syntax tree created by cho
 * You can use all the cool new f
 * and even more. Enjoy!
 */

let tips = [
  "Click on any AST node with a

  "Hovering over a node highlight
   corresponding location in the

  "Shift click on an AST node to
];

function printTips() {
  tips.forEach((tip, i) => consol
}
```

☑ Autofocus ☑ Hide methods ☐ Hide empty keys ☐ Hide location data ☐ Hide type keys

```
- Program  {
     type: "Program"
     start: 0
     end: 481
   - body:  [
      + VariableDeclaration {type, start, end, declarations, kind}
      + FunctionDeclaration {type, start, end, id, expression, ... +4}
     ]
     sourceType: "module"
  }
```

Question Find (press in the code to the element you want to find) in the AST for the example above the class name (see the picture below). What fields are stored in the node of this tree? Why do you think?


AST Explorer
Snippet
Java
java-parser
Transform
default
Parser: java-parser-0.6.0
Tree
JSON
56ms
Built with React, Babel, Font Awesome, CodeMirror, Express, and webpack | GitHub | Build: a477e48

```
public class Program {
    public static void main(String[] args) {
        int x = 100;
        if (x > 0) {
            System.out.println("Greater than zero");
        }
        else {
            System.out.println("Less than or equal to zero");
        }
    }
}
```

☑ Autofocus ☑ Hide methods ☑ Hide empty keys ☑ Hide location data ☑ Hide type keys

```
- normalClassDeclaration:  [
  - normalClassDeclaration  {
    - children:  {
      + Class: [1 element]
      - typeIdentifier:  [
        - typeIdentifier  {
          - children:  {
            - Identifier:  [
              -  {
                   image: "Program"
                   startOffset: 13
                   endOffset: 19
                   startLine: 1
                   endLine: 1
                   startColumn: 14
                   endColumn: 20
                   tokenTypeIdx: 3
                 }
              ]
            }
          }
        ]
      + classBody: [1 element]
      }
    }
  ]
}
```

Question 3 Find the “if” statement in the AST for the example above. Add the code to repeat the situation with the if-else clause from the first example. Draw AST view for this code according to AST on the site. Send the screenshot like the picture above but for “if” and send your drawing of AST.

Now let's look at a simple AST tree printer program. The tree will be set right in the code using a small domain-specific language, which we will define right in the program.

Question examine the file `printer0.c`.

Question 4 Add the code in `main()` to display the following expressions (parentheses are enable in the output). Send these lines of code through the form.

- 999 + 728
- 4+2*9

Question 5 Extend this example by adding the vertex type AST for division and a printer for it. So the program can output the example: (3+5) * (9 / 7). Send the program code. (Check it can be compiled first).