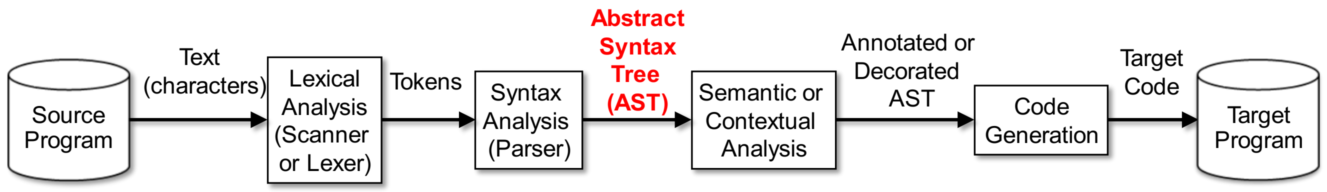
Programming Language Design
Department of Computer Science
Dr. Francisco Ortin
francisco.ortin@mtu.ie

Notice: Please, read the language specification file (language-specification.pdf) before reading this document.

Laboratory 2 – Design of an Abstract Syntax Tree (AST)

As mentioned in lectures, compilers follow the Pipes and Filters architectural pattern. In our compiler, we have the following phases:


Source Program
Text (characters)
Lexical Analysis (Scanner or Lexer)
Tokens
Syntax Analysis (Parser)
Abstract Syntax Tree (AST)
Semantic or Contextual Analysis
Annotated or Decorated AST
Code Generation
Target Code
Target Program

A key data structure is the AST, returned by the parser, decorated by the semantic analyzer, and traversed to generate code in the code generation phase. Therefore, the AST of a programming language must be carefully designed, fulfilling the following requirements:

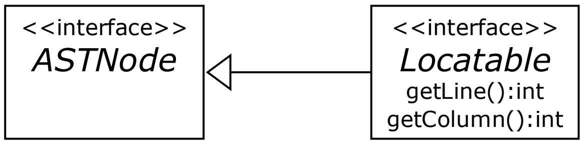
- The AST should represent the structure of an input program in memory.
- Any syntactically valid program could be represented with an AST.
- The structure and information to be stored in the AST should be reduced to the minimum. Thus, symbols such as semicolon (;), comma (,) or parenthesis should not be included in the AST since their only purpose is the unambiguous parsing of the input.

The objective of this laboratory is to design and implement the AST for the language defined in the language-specification file. This is a software design activity.

First Steps

After analyzing the language description (language-specification.pdf) and the sample input.txt file, we should be able to design the classes in the AST using UML.

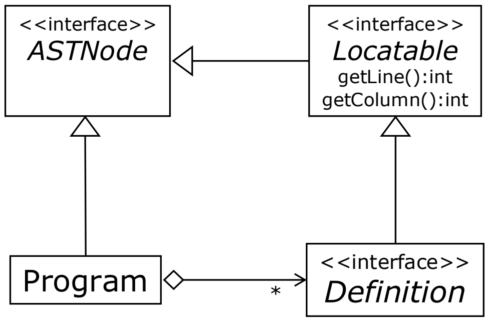
First, all the nodes in the AST must implement a common interface that we will use later to perform the AST traversal. We name that interface `ASTNode`. Next, most AST nodes will provide their location in the source file by implementing the `Locatable` interface. Since our language only supports one-file programs, we just need to provide the line and column numbers for each node. This information will be used when the compiler prompts an error in the source program. The position of the wrong input will be taken from the `Locatable` AST node and included in the error message.



The language specification says:

A program is defined as a sequence of variable and function definitions. They can be intermixed.

So, we know a program is a collection of something called "definition", that generalizes both variable and function definitions. We can design that generalization using polymorphism, so we have:

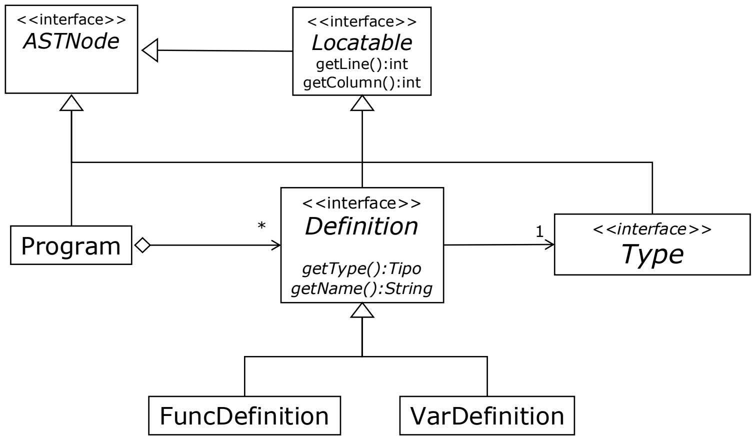


The interface `Definition` generalizes variable and function definitions. In this way, a program is a collection of definitions (the aggregation relationship between `Program` and `Definition` in the previous diagram). Notice that all the definitions have a position in the input file, but `Program` has no localization.

Then, we check the language specification for the two specific definitions (variable and function definitions):

The syntax of a variable definition is a type followed by an identifier and one final ";" character. Functions are defined by specifying the return type, the function identifier, and a list of comma-separated parameters between (and). The return type and parameter types must be built-in (i.e., no arrays).

Therefore, both kinds of definitions have one type (the type of the variable and the function type[1]). They also have the name of the variable (or function) declared. Therefore, we could extend our design in the following way:



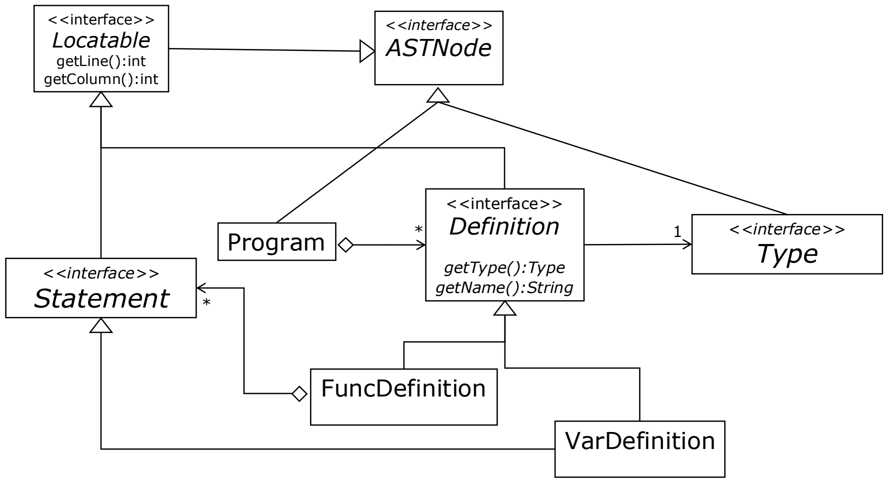
[1] Please, notice that a function type is not the type returned by the function. For example, in the function `double f(int a)`, the type of `f` is not `double`; it is a function type that receives an `int` and returns a `double`.

`Type` is an interface that generalizes any type, and must be implemented by the different types in the language, including function types.

The following text describes function bodies:

The function body goes between { and }. The bodies of functions are sequences of local variable definitions, followed by sequences of statements. Both must end with the ";" character.

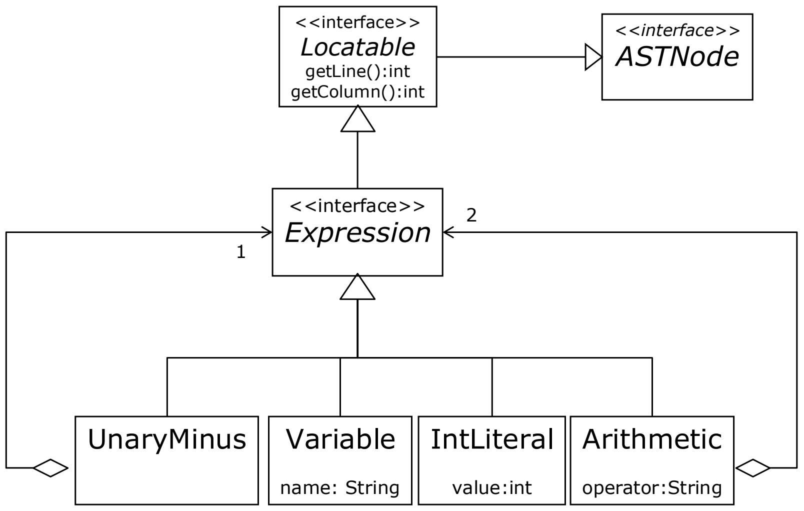
Thus, a function definition must collect a sequence of variable definitions and a sequence of statements. We model the general concept of statement as an interface. We have two options for modeling function bodies. The first option consists in creating two association relationships from `FuncDefinition`: one pointing to `VarDefinition` and another one pointing to `Statement`. The second option is to identify variable definitions as statements, and use one single collection (association):



This second option is more general and allows the definition of local variables anywhere inside the function body. That is not what the language specifies, but the language grammar (the parser) will forbid that construction in the syntax analysis. Both alternatives are correct to implement C--, but the second one is closer to most existing languages.

Go Ahead

You should go ahead with the design and finish the UML class diagram of the AST. The design of expressions and types will make use of the *Composite* design pattern. For instance, variables (identifiers), integer literals, binary arithmetic operations (e.g., arithmetic expressions created with `+`, `-`, `*`, and `/`) and the unary – operator could be designed the following way, as an instance of the *Composite* design pattern:



In the previous class diagram:

– `Variable` represents identifiers (e.g., `a`, `b`, `var1`, `another_var`…). Their names are stored in the `name` field.
– `IntLiteral` represents integer literals without sign (e.g., `1`, `2`, `33`, `1234`…).
– `UnaryMinus` represents unary expressions created with the `-` operator (e.g., `-a`, `-3`, `-(a+b)`, `-(-b)`…).
– `Arithmetic` represents binary arithmetic expressions, created with the `+`, `-`, `*` and `/` operators (e.g., `1+b`, `a*2`, `7-(c+8)`, `a/-3`…).

Now, you should finish the design for all the *expressions* in the language.

After expressions, you should design all the *statements* and *types* in the language (remember that C-- provides function types). The objective of this lab is to finish the design of the AST for the whole language (follow the description of the language in language-specification.pdf). Use any tool to create the UML class diagrams (you are going to use those diagrams throughout the module, since the AST is the most important data structure in your compiler)—an example free, online and very basic tool you can use is draw.io.

Once you finish the design, implement it in Java. Create a new lab02 Java project with an `ast` package. Implement your AST design (interfaces and classes) in the `ast` package. The ASTTest.java file provided shows an example creation of an AST for a tiny input program. It is displayed with the visual Introspector[2] tool already used in lab 01. Introspector shows the structure of any Java object in memory. Thus, it is useful to show the structure of ASTs. You can take a look at the IntrospectorDemo.java file for an example use of that tool.

You do not need to add the `ast` package to the mini-compiler (create a new Java project). However, you can take a look at the `ast` and `ast.types` packages in the mini-compiler to see a working example (it is quite helpful). In lab05, you will use your AST implementation, once you have implemented the lexer (lab03) and the parser (lab04) for the C-- language.

[2] Please, download the last version of Introspector from https://github.com/francisco-ortin/Introspector

Should you have any questions, please use the discussion forum to ask them to the rest of the students and lectures in the module. If you want them to be private, please send me a message through Canvas Inbox.