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

Laboratory 6 – AST Traversal

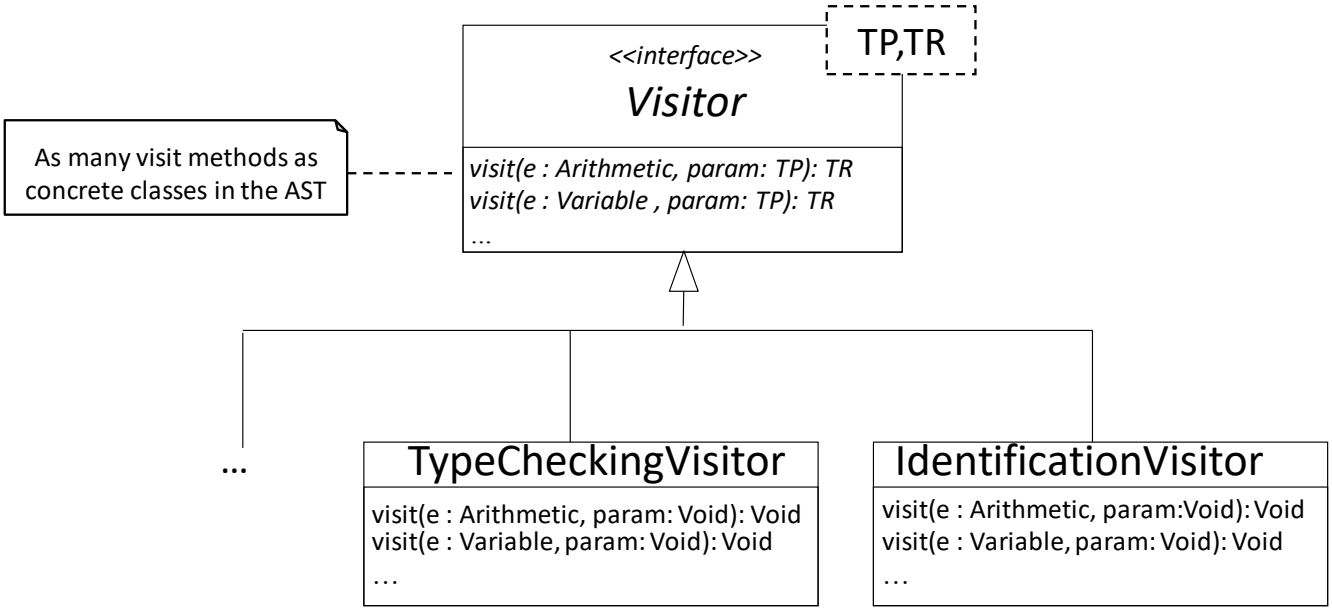
The objective of this laboratory is to define a versatile mechanism to traverse ASTs. AST traversal is necessary for semantic analysis and code generation and optimization. In this lab, we will traverse the AST for a basic scenario: checking l-value expressions.

The Visitor design pattern

The Interpreter design pattern [1] allows traversing tree structures. However, the traversal methods are added to the tree nodes, mixing the tree abstraction with its traversals. When building a compiler, the AST will be traversed in many different ways, so it would be helpful to:

- Separate the traversals from the tree nodes.
- Add new traversals without modifying the tree.

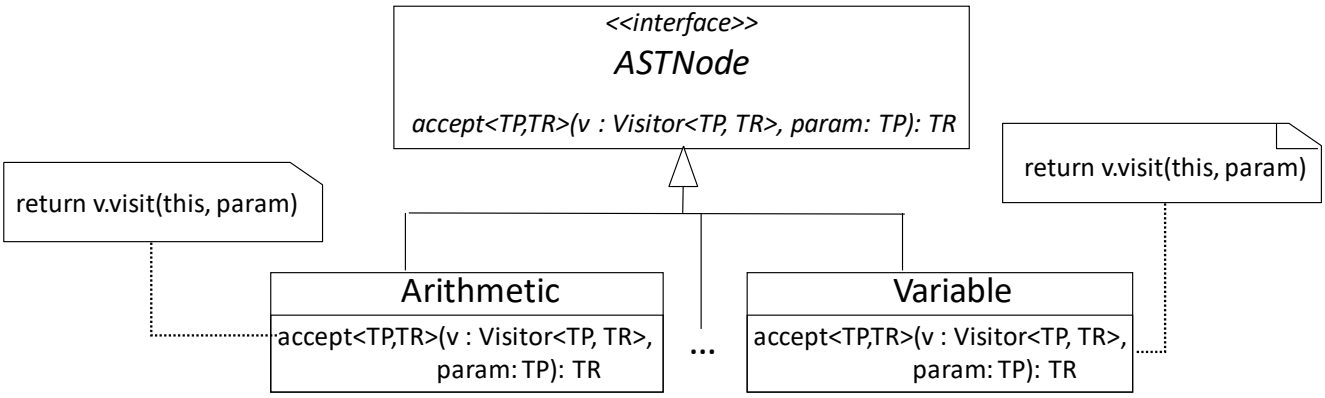
The Visitor design pattern [1] provides these two objectives. Traversals are represented with Visitor abstractions (e.g., `TypeCheckingVisitor` and `IdentificationVisitor`), generalized with the `Visitor` interface:



Each `visit` method defines the concrete traversal of a concrete node for a particular purpose (e.g., type checking or variable identification). Abstract visitor classes can be used to define default traversal implementations.

The class diagram above generalizes the Visitor design pattern in [1] with generic `TP` and `TR` types. `TP` allows passing one parameter of any type to the traversal algorithm. `TR` abstracts the return type. The particular traversals (derived classes) instantiate these two generic types depending on their particular traversal requirements (`Void` if no type is required).

Since polymorphic types such as `Statement`, `Expression` or `ASTNode` are too general, the `visit` method cannot be implemented for those types. This is because the actual structure of a statement, expression or AST node is unknown, since they are very general abstractions. However, we will need to traverse those types of nodes. For instance, when traversing an assignment, we will need to traverse its two child nodes, which are both `Expression` (polymorphic) types. To this end, the Visitor design pattern defines a mechanism to traverse *any* node in the AST (including the general ones): a polymorphic `accept` method, implemented in all the AST nodes. Thus, `accept` (not `visit`) is the method that must be used to traverse trees (remember not to invoke `visit`; invoke `accept` instead).



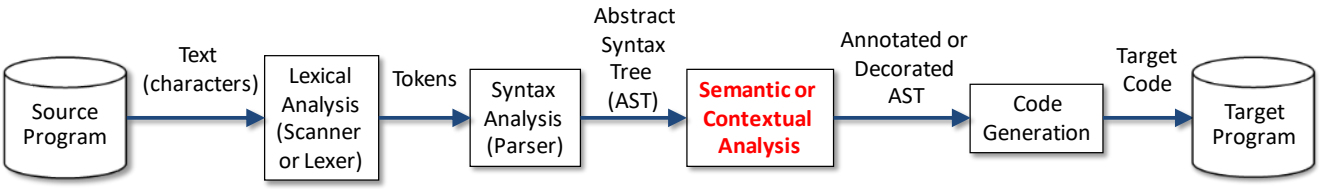
The outcome is that the `accept` method provides a kind of double dispatch in Java, a programming language feature that is commonly provided as multiple dispatch (i.e., multi-methods). Java does not support multiple dispatch.

Activity

The semantic analysis phase of our compiler implements two visitors: `TypeCheckingVisitor` and `IdentificationVisitor`. In this lab, we only implement *a part* of `TypeCheckingVisitor`. The purpose of this lab is to just develop our first traversal –it will be extended in forthcoming labs.

One of the responsibilities of `TypeCheckingVisitor` is to check l-value expressions in the assignment and read statements. An l-value is an expression that can be placed on the left-hand side of assignments. That is, an expression that has a modifiable memory address. The expressions `a` and `v[i+3]` are l-values; `34` and `a+b` are not.

Your compiler must check that expressions on the left-hand side of assignments *are* l-values. This condition must also be checked in read statements.



Go ahead

Include the Visitor design pattern in your compiler.

Create a specific `TypeCheckingVisitor` in your semantic analyzer to check l-values.

For the two incorrect files provided, your compiler must show an informative semantic error, printing the line and column. input.txt must be compiled without any error.

Semantic errors should be represented with instances of `ErrorType`, a class implementing the `Type` interface. All the instances of `ErrorType` must be collected by an error manager. Input programs may have more than one semantic error. After semantic analysis, all the semantic errors must be shown to the programmer. For each error, a descriptive message and its line and column must be displayed. This functionality is available in the mini compiler from lab 01.

References

[1] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1995.