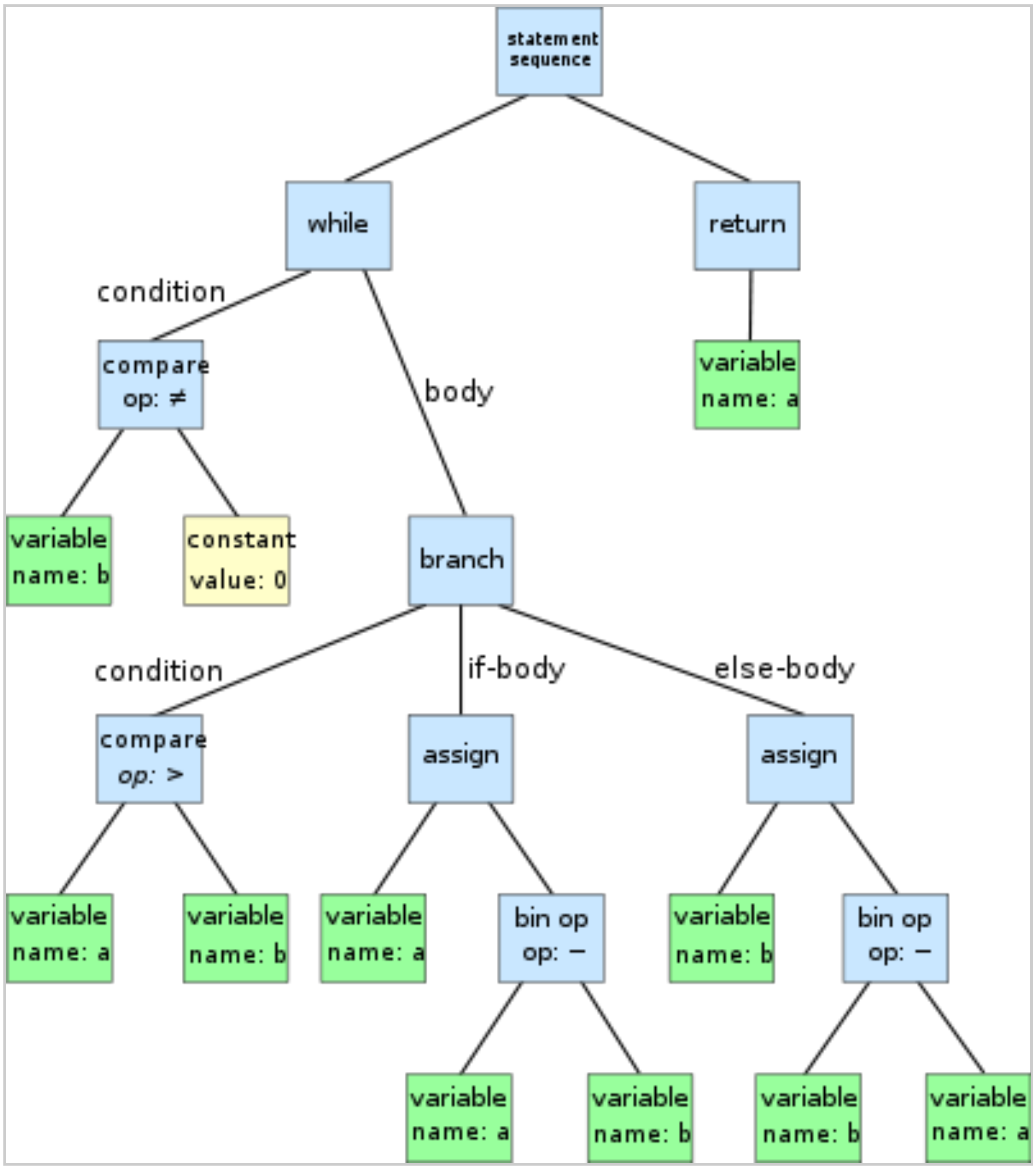
Abstract syntax tree

From Wikipedia, the free encyclopedia

In computer science, an **abstract syntax tree** (**AST**), or just **syntax tree**, is a tree representation of the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a construct occurring in the source code. The syntax is "abstract" in not representing every detail appearing in the real syntax. For instance, grouping parentheses are implicit in the tree structure, and a syntactic construct like an if-condition-then expression may be denoted by means of a single node with three branches.

This distinguishes abstract syntax trees from concrete syntax trees, traditionally designated parse trees, which are often built by a parser during the source code translation and compiling process. Once built, additional information is added to the AST by means of subsequent processing, e.g., contextual analysis.

Abstract syntax trees are also used in program analysis and program transformation systems.



An abstract syntax tree for the following code for the Euclidean algorithm:

```
while b ≠ 0

    if a > b

        a := a − b

    else

        b := b − a

return a
```

Contents

Application in compilers

Abstract syntax trees are data structures widely used in compilers, due to their property of representing the structure of program code. An AST is usually the result of the syntax analysis phase of a compiler. It often serves as an intermediate representation of the program through several stages that the compiler requires, and has a strong impact on the final output of the compiler.

Motivation

Being the product of the syntax analysis phase of a compiler, the AST has several properties that are invaluable to the further steps of the compilation process.

- Compared to the source code, an AST does not include certain elements, such as inessential punctuation and delimiters (braces, semicolons, parentheses, etc.).
- A more important difference is that the AST can be edited and enhanced with information such as properties and annotations for every element it contains. Such editing and annotation is impossible with the source code of a program, since it would imply changing it.
- At the same time, an AST usually contains extra information about the program, due to the consecutive stages of analysis by the compiler, an example being the position of an element in the source code. This information may be used to notify the user of the location of an error in the code.

ASTs are needed because of the inherent nature of programming languages and their documentation. Languages are often ambiguous by nature. In order to avoid this ambiguity, programming languages are often specified as a context free grammar (CFG). However, there are often aspects of programming languages that a CFG can't express, but are part of the language and are documented in its specification. These are details that require a context to determine their validity and behaviour. For example, if a language allows new types to be declared, a CFG cannot predict the names of such types nor the way in which they should be used. Even if a language has a predefined set of types, enforcing proper usage usually requires some context. Another example is duck typing, where the type of an element can change depending on context. Operator overloading is yet another case where correct usage and final function are determined based on the context. Java provides an excellent example, where the '+' operator is both numerical addition and concatenation of strings.

Although there are other data structures involved in the inner workings of a compiler, the AST performs a unique function. During the first stage, the syntax analysis stage, a compiler produces a parse tree. This parse tree can be used to perform almost all functions of a compiler by means of syntax-directed translation. Although this method can lead to a more efficient compiler, it goes against the software engineering principles of writing and maintaining programs. Another advantage that the AST has over a parse tree is the size, particularly the smaller height of the AST and the smaller number of elements.

Design

The design of an AST is often closely linked with the design of a compiler and its expected features.

Core requirements include the following:

- Variable types must be preserved, as well as the location of each declaration in source code.
- The order of executable statements must be explicitly represented and well defined.
- Left and right components of binary operations must be stored and correctly identified.
- Identifiers and their assigned values must be stored for assignment statements.

These requirements can be used to design the data structure for the AST.

Some operations will always require two elements, such as the two terms for addition. However, some language constructs require an arbitrarily large number of children, such as argument lists passed to programs from the command shell. As a result, an AST used to represent code written in such a language has to also be flexible enough to allow for quick addition of an unknown quantity of children.

Another major design requirement for an AST is that it should be possible to unparse an AST into source code form. The source code produced should be sufficiently similar to the original in appearance and identical in execution, upon recompilation.

Design patterns

Due to the complexity of the requirements for an AST and the overall complexity of a compiler, it is beneficial to apply sound software development principles. One of these is to use proven design patterns to enhance modularity and ease of development.

Different operations don't necessarily have different types, so it is important to have a sound node class hierarchy. This is crucial in the creation and the modification of the AST as the compiler progresses.

Because the compiler traverses the tree several times to determine syntactic correctness, it is important to make traversing the tree a simple operation. The compiler executes a specific set of operations, depending on the type of each node, upon reaching it, so it often makes sense to use the visitor pattern.

Usage

The AST is used intensively during semantic analysis, where the compiler checks for correct usage of the elements of the program and the language. The compiler also generates symbol tables based on the AST during semantic analysis. A complete traversal of the tree allows verification of the correctness of the program.

After verifying correctness, the AST serves as the base for code generation. The AST is often used to generate the 'intermediate representation' '(IR)', sometimes called an intermediate language, for the code generation.

See also

- Abstract semantic graph (ASG)
- Composite pattern
- Document Object Model (DOM)
- Extended Backus–Naur Form
- Lisp, a family of languages written in trees, with macros to manipulate code trees at compile time
- Semantic resolution tree (SRT)
- Shunting yard algorithm

- Symbol table
- TreeDL
- Term graph

References

- This article is based on material taken from the Free On-line Dictionary of Computing prior to 1 November 2008 and incorporated under the "relicensing" terms of the GFDL, version 1.3 or later.

Further reading

- Jones, Joel. "Abstract Syntax Tree Implementation Idioms" (PDF). (overview of AST implementation in various language families)
- Neamtiu, Iulian; Foster, Jeffrey S.; Hicks, Michael (May 17, 2005). *Understanding Source Code Evolution Using Abstract Syntax Tree Matching*. MSR'05. Saint Louis, Missouri: ACM. CiteSeerX: 10.1.1.88.5815. delete character in `|publisher=` at position 17 (help); delete character in `|conference=` at position 17 (help)
- Baxter, Ira D.; Yahin, Andrew; Moura, Leonardo; Sant' Anna, Marcelo; Bier, Lorraine (November 16–19, 1998). *Clone Detection Using Abstract Syntax Trees* (PDF). *Proceedings of ICSM'98* (Bethesda, Maryland: IEEE). delete character in `|work=` at position 32 (help)
- Fluri, Beat; Würsch, Michael; Pinzger, Martin; Gall, Harald C. "Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction" (PDF).
- Würsch, Michael. *Improving Abstract Syntax Tree based Source Code Change Detection* (Diploma thesis).
- Lucas, Jason. "Thoughts on the Visual C++ Abstract Syntax Tree (AST)".

External links

- AST View (http://www.eclipse.org/jdt/ui/astview/index.php): an Eclipse plugin to visualize a Java abstract syntax tree
- "Good information about the Eclipse AST and Java Code Manipulation". *eclipse.org*.
- PMD (https://sourceforge.net/projects/pmd/) on SourceForge.net: uses AST representation to control code source quality
- "CAST representation". *cs.utah.edu*.
- eli project (http://eli-project.sourceforge.net/elionline/idem_3.html): Abstract Syntax Tree Unparsing
- "Abstract Syntax Tree Metamodel Standard" (PDF).
- "Architecture‑Driven Modernization — ADM: Abstract Syntax Tree Metmodel — ASTM". (OMG standard).

Retrieved from "https://en.wikipedia.org/w/index.php?title=Abstract_syntax_tree&oldid=692162487"

Categories: Trees (data structures) | Formal languages
