Programming Languages: Lecture 16 Introduction to Attributes and Attribute Grammars

Rishabh Dhiman

4 February 2022

The CFG of a PL forms a framework for designing the back-end of the compiler in parts,

- The concrete parse tree itself never actually needs to be generated.
- It is conceptually helpful to think of it as the result of parsing.

Unlike the usual D&C method,

- here the division of the backend is in terms of the non-terminals, and
- the grammatical rules of the language,
- "divide" is more nuanced and the individual parts are also connected by rules and have overlaps.

The following are broadly classified as attributes (of some kind) that may have to be generated,

- scope information,
- type information,
- relocatable memory address,
- abstract syntax trees,
- target code.

Practically everything done after parsing is an attribute.

All of this came from Knuth's TeX compilation, it turned out to be general enough that it became a structure of a general compiler.

1 Abstract Syntax Trees

The construction of ASTs frmom concrete parse trees is an example of a transforamtion that can be perfuromed using a synax-directed definition that has no side-effects.

An attribute grammar is a formal way to define semantic rules and context-sensitive aspects of the language. Each production of the grammar is associated with a set of values or semantic rules. These values and semantic rules are collectively referred to as attributes.

2 Abstract Syntax

Shift-reduce parsing produces a concrete syntax tree from the rightmost derivation. The syntax tree is concrete in the sense that

- It contains a lot of redundant symbols that are important or useful only during the parsing stage.
- brackets of various kinds.

Every PL construct is possibly an operator.
A program regarded as an AST is simply a composition of operators.

2.1 Imperative Approach

We use an attribute grammar rules to construct the AST from the parse tree.

We define two procedures first

- `makeLeaf(literal)` makes a leaf node from the given literal.
- `makeBinaryNode(opr, opd1, opd2)` createts a node with label *opr* (with fields which point to *opd1* and *opd2*) and returns a pointer or reference to the newly created node.

Now we may associate a synthesized attribute called *ptr* with each terminal and nonterminal symbol which points to the root of the subtree created for it.

2.2 Functional Approach

We use attribute grammar rules to construct the abstract syntax tree functionally from the parse tree.

2.3 Alternative Functional

In languages which support algebraic (abstract) datatypes, the functions `makeLeaf` and `makeBinaryNode` may be replaced by the constructors of an appropriate recursively defined datatype AST.

Synthesized and inherited attributes to make *information flow* from one part of the grammar to another

3 Attributes

The value of an attribute at a parse-tree node is defined by the semantic rule associated with the production used at that node.

4 The Structure of a Compiler

- Syntax-Directed Definitions (SDD)
- Divide and Conquer
- Glue code

4.1 Syntax-Directed Definitions

Syntax-Directed definitions are high-level specifications which specify the eval- uation of various attributes and procedures.

They hide various implementation details and free the compiler writer from explicitly defining the order in which translation, transformations, and code generation take place.

5 Kinds of Attributes

- **Synthesized Attributes.** A *synthesized* attribute is one whose value depends upon the values of its immediate children in the concrete parse tree.

  An SDD that usese only synthesized attributes is called an *S-attributed* definition.

- **Inherited Attributes.** An inherited attributed is one whose value depends upon the values of the attributes of its parents or siblings in the parse tree.

  Inherited attributes are convenient for expressing the dependence of a language construct on the *context* in which it appears.