Theme 3
Semantic Analysis
Attribute grammars, symbol table

Compiladores, 2º semestre 2022-2023

Miguel Oliveira e Silva, Artur Pereira, DETI, University of Aveiro

Contents

1 Semantic Analysis: Structure of a Compiler **2**
1.1 Syntax driven evaluation 2
1.2 Static or dynamic detection 2

2 Type system **3**

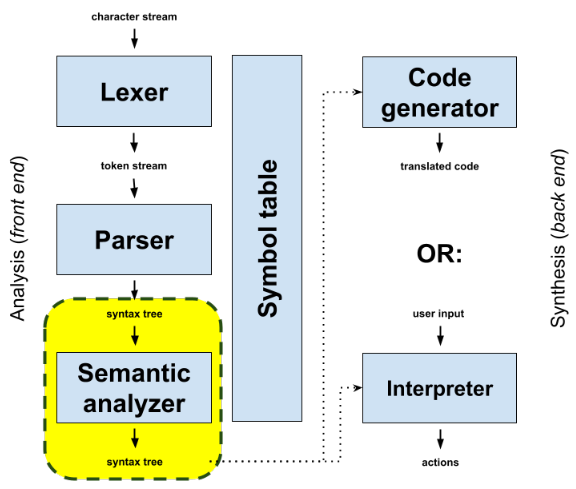
3 Attribute grammars **3**
3.1 Local dependency: attribute classification 4
3.2 ANTLR4: Declaration of attributes associated with the syntax tree 5

4 Symbol Table **6**
4.1 Grouping symbols in contexts 7

5 Context-restricted Instructions **8**

1 Semantic Analysis: Structure of a Compiler

- Let's now analyze the semantic analysis phase in more detail:



- When processing a language, semantic analysis must ensure, as much as possible, that the source program makes sense (by the rules defined in the language).
- Common semantic errors:
  - Variable/function not defined;
  - Incompatible types (e.g. assigning a real number to an integer variable, or using a non-boolean expression in the conditions of a conditional statement);
  - Define instruction in wrong context (e.g. use in `Java` the instruction `break` outside a loop or `switch`).
  - Meaningless application of instruction (e.g. importing a non-existent *package* in `Java`).
- In some cases, these errors can still be evaluated during parsing; In other cases, it is only possible do this check after a successful parsing, making use of the information extracted from that parsing.

1.1 Syntax driven evaluation

- In language processing, semantic evaluation can be done by associating information and actions with rules syntax of the grammar (i.e. to *parse tree* nodes).
- This procedure is called *syntax driven evaluation*.
- For example, in a grammar for arithmetic expressions we can associate a variable with the node of the tree with the type of expression, and actions that allow checking its correctness (and not allowing, for example, trying to add a boolean with an integer).
- In ANTLR4, the association of attributes and actions to the syntax tree can be done during the analysis itself syntax, and/or later using *visitors* and/or *listeners*.

1.2 Static or dynamic detection

- The verification of each semantic property of a language can be done in two different times:
  - In *dynamic time*: that is, during *runtime*;
  - At *static time*: that is, during *compile time*.
- Only in compilers do static checks of semantic properties make sense.
- In interpreters, the analysis and synthesis phases of the language are both done at runtime, so checks are always dynamic.

- Static checking has the advantage of guaranteeing, at runtime, that certain errors will never occur (by dispensing with the need to proceed with its debugging and testing).

2 Type system

- The type system of a programming language is a formal logical system, with a set of semantic rules, that by associating a property (type) to language entities (expressions, variables, methods, etc.) allows the detection of an important class of semantic errors: *type errors*.
- Type error checking is applicable for the following operations:
  - Value assignment: $v = e$
  - Applying operators: $e_1 + e_2$ (for example)
  - Function invocation: $f(a)$
  - Use of classes/structures: $o.m(a)$ or $data.field$
- Other operations, such as using arrays, may also involve type checking. However, we can consider that operations on arrays are value assignments and application of special methods.
- Any of these operations is said to be valid when there is *conformity* between the properties of the entities involved.
- The conformance indicates whether a T_2 type can be used where a T_1 type is expected. This is what happens when $T_1 = T_2$.
  - Value Assignment ($v = e$).
    The type of e must conform to the type of v
  - Application of operators ($e_1 + e_2$).
    There is an operator $+$ applicable to the types of e_1 and e_2
  - Function invocation ($f(a)$).
    There is a global function f that accepts arguments a conform to the declared formal arguments of that function.
  - Using classes/structures ($o.m(a)$ ou $data.field$).
    There is a method m in the class corresponding to the object o, which accepts arguments a according to the declared formal arguments of that method; and there is a field $field$ in the structure/class of $data$.

3 Attribute grammars

- We have already seen that attributing meaning to the source code of a language requires not only syntactic correction (ensured by context-independent grammars) as well as semantic correction.
- In this sense, it is very convenient to have access to all the information generated by the syntactic analysis, i.e. to the syntax tree, and be able to associate new information to the respective nodes.
- This is the purpose of the *attribute grammar*:
  - Each language grammar symbol (terminal or non-terminal) can be associated with a set of zero or more *attributes*.
  - An attribute can be a number, a word, a type, ...
  - The calculation of each attribute has to be done taking into account the dependence of the necessary information for its value.
- Among the different types of attributes, there are some whose value depends only on their syntactic neighborhood.
  - One such example is the value of an arithmetic expression (which moreover depends only on the node itself and therefore eventually from descendant nodes).

- There are also attributes that (may) depend on remote information.
  - This is the case, for example, for the data type of an expression involving the use of a variable or invocation of a method.

3.1 Local dependency: attribute classification

- Attributes can be classified in two ways, depending on the dependencies that apply to them:
  1. They are said to be *synthesized*, if their value depends only on descendant nodes (i.e. if their value depends only on the existing symbols in the respective production body).
  2. They are said to be *inherited*, whether it depends on sibling or ascendant nodes.

$$\text{parent} \rightarrow \text{child}\,\text{child}$$

$$\begin{tikzcd}
& \text{parent} \arrow[dl, "\textit{inherited}"'] & \\
\text{child} \arrow[rr, "\textit{inherited}"'] & & \text{child} \arrow[ul, "\textit{synthesized}"']
\end{tikzcd}$$

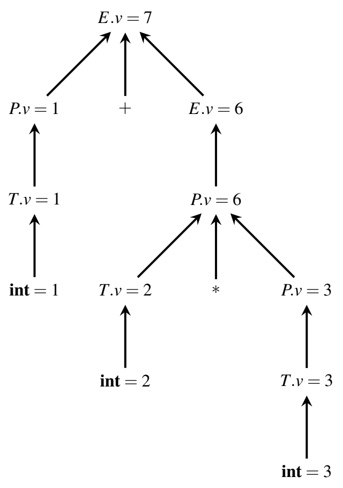
- You can formally designate attributes by annotating them with an arrow in the direction of dependency (upwards for synthesized attributes and downwards for inherited ones).

Example local dependency: arithmetic expression

- Consider the following grammar:

$$\begin{aligned} E &\rightarrow P + E \mid P \\ P &\rightarrow T * P \mid T \\ T &\rightarrow (E) \mid \textbf{int} \end{aligned}$$

- If we want to set a `v` attribute to the value of the expression, we have an example of a synthesized attribute.
- For example, for the input — $1+2*3$ — we have the following annotated parse tree:



$$\begin{aligned} E &\rightarrow P + E \mid P \\ P &\rightarrow T * P \mid T \\ T &\rightarrow (E) \mid \textbf{int} \end{aligned}$$

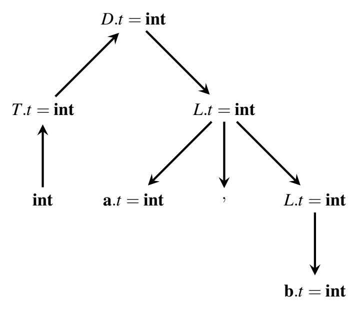
Production	**Semantic rule**
$E_1 \rightarrow P + E_2$	$E_1.v = P.v + E_2.v$
$E \rightarrow P$	$E.v = P.v$
$P_1 \rightarrow T * P_2$	$P_1.v = T.v * P_2.v$
$P \rightarrow T$	$P.v = T.v$
$T \rightarrow (E)$	$T.v = E.v$
$T \rightarrow \textbf{int}$	$T.v = \textbf{int}.value$

Example local dependency: declaration

- Consider the following grammar:

$$\begin{aligned} D &\rightarrow T\,L \\ T &\rightarrow \textbf{int} \mid \textbf{real} \\ L &\rightarrow \textbf{id}\,,L \mid \textbf{id} \end{aligned}$$

- If we want to define a `t` attribute to indicate the type of each **id** variable, we have an example of an inherited attribute.
- For example, for the input — **int** a, b — we have the following annotated parse tree:



$$\begin{aligned} D &\rightarrow T\,L \\ T &\rightarrow \textbf{int} \mid \textbf{real} \\ L &\rightarrow \textbf{id}\,,L \mid \textbf{id} \end{aligned}$$

Production	**Semantic rule**
$D \rightarrow T\,L$	$D.t = T.t$ $L.t = T.t$
$T \rightarrow \textbf{int}$	$T.t = \textbf{int}$
$T \rightarrow \textbf{real}$	$T.t = \textbf{real}$
$L_1 \rightarrow \textbf{id}\,,L_2$	$\textbf{id}.t = L_1.t$ $L_2.t = L_1.t$
$L \rightarrow \textbf{id}$	$\textbf{id}.t = L.t$

3.2 ANTLR4: Declaration of attributes associated with the syntax tree

- We can declare attributes in different ways:
  1. Directly in the context-independent grammar using arguments and results of syntactic rules;

```
expr[String type] returns[int value]: // type not used
      e1=expr '+' e2=expr
      {$value = $e1.value + $e2.value;}        #ExprAdd
   |  INT
      {$value = Integer.parseInt($INT.text);} #ExprInt
   ;
```

  2. Indirectly making use of the associative array `ParseTreeProperty`:

```
protected ParseTreeProperty<Integer> value =
   new ParseTreeProperty<>();
...
@Override public void exitInt(ExprParser.IntContext ctx){
   value.put(ctx, Integer.parseInt(ctx.INT().getText()));
}
...
@Override public void exitAdd(ExprParser.AddContext ctx){
   int left = value.get(ctx.e1);
   int right = value.get(ctx.e2);
   value.put(ctx, left + right);
}
```

- We can also use the result of the `visit` methods.

ANTLR4: Attribute declaration `ParseTreeProperty`

- This array has nodes of the syntax tree as key, and allows simulating both arguments and results, of rules.
- The difference is where its value is assigned and accessed.
- To simulate the passage of *arguments* just assign the value *before* traversing the respective node (in *listeners* usually in `enter...` methods), access being made in the *own* node.
- To simulate *results*, proceed as in the given example (i.e. assign the value in the *own* node, and access it in the *upward* nodes).

Attribute grammars in ANTLR4: synthesis

- We can associate three types of information to syntactic rules:
    1. Information originating from rules used in the body of the rule (synthesized attributes);
    2. Information from rules that use this rule in their body (inherited attributes);
    3. Information local to the rule.
- In `ANTLR4` the direct use of all these types of attributes is very simple and intuitive:
    1. Synthesized attributes: result of rules;
    2. Inherited attributes: rule arguments;
    3. Local attributes.
- Alternatively, we can use the associative array `ParseTreeProperty` (which is justified only for the first two, since for the third we can use local variables to the respective method); or the result of the `visit` methods (in the case of using *visitors*) for synthesized attributes.

4 Symbol Table

- The attribute grammar is well suited to dealing with attributes with local dependency.
- However, we may have information whose origin has no direct dependence on the syntax tree (for example, multiple appearances of a variable), or that may even reside in the processing of other source code (for example, names of classes defined in another file).
- Thus, whenever the language uses symbols to represent program entities – such as: variables, functions, registers, classes, etc. – it becomes necessary to associate to the symbol id (usually an identifier) its definition (symbol category, associated data type).
- This is why the *symbol table* exists.
- The symbol table is an associative array, where the key is the name of the symbol, and the element an object that defines the symbol.
- Symbol tables can be global or local (for example, to a block of code or a function).

- The information associated with each symbol depends on the type of language defined, as well as whether we are in the presence of an interpreter or a compiler.
- Examples of these properties are:
    - **Name**: symbol name (associative array key);
    - **Category**: what does the symbol represent, class, method, object variable, local variable, etc.;
    - **Type**: symbol data type;
    - **Value**: value associated with the symbol (only in case of interpreters).
    - **Visibility**: restriction on symbol access (for languages with encapsulation).

Symbol Table: Implementation

- In an object-oriented approach we can define the abstract class `Symbol`:

```
public abstract class Symbol {
    public Symbol(String name, Type type) { ... }
    public String name() { ... }
    public Type type() { ... }
}
```

- We can now define a variable:

```
public class VariableSymbol extends Symbol {
    public VariableSymbol(String name, Type type) {
       super(name, type);
    }
}
```

- The `Type` class allows identification and verification of conformity between types:

```
public abstract class Type {
   protected Type(String name) { ... }
   public String name() { ... }
   public boolean subtype(Type other) {
      assert other != null;
      return name.equals(other.name());
   }
}
```

- We can now implement specific types:

```
public class RealType extends Type {
   public RealType() { super("real"); }
}

public class IntegerType extends Type {
   public IntegerType() { super("integer"); }

   public boolean subtype(Type other) {
      return super.subtype(other) ||
             other.name().equals("real");
   }
}
```

4.1 Grouping symbols in contexts

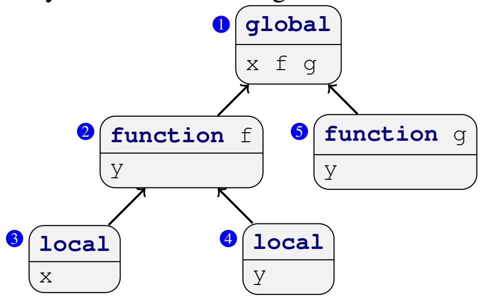
- If the language is simple, containing a single symbol definition context, then the lifetime of symbols is linked to the lifetime of the program, and a single symbol table is sufficient.
- However, if we have the possibility to define symbols in different contexts, so we need to solve the problem of symbols having lifetimes (and/or visibility) that depend on the context within the program.

- Consider the following code as an example (in `C` language):

```
❶ // start of global scope
  int x;          // define variable x in global scope
❷ void f() {      // define function f in global scope
    int y;        // define variable y in local scope of f
❸   { int x; }    // define variable x in nested local scope
❹   { int y; }    // define variable y in another nested local scope
  }
❺ void g() {      // define function g in global scope
    int y;        // define variable y in local scope of g
  }
...
```

- The numbering identifies the different symbol contexts.
- A very important aspect is that contexts can be defined within other contexts.
- Thus the context ❷ is defined inside the context ❶; and in turn the context ❸ is defined inside ❷.

- In ❹ the symbol `x` is defined in ❶.
- To adequately represent this information, the different symbol tables are structured in a tree where each node represents a stack of symbol tables starting at that node to the root (global symbol table).



- Depending on where we are in the program. we have a stack of symbol tables defined to resolve the symbols.
- Symbol names may be repeated, as defined in the nearest table (in stack order).
- In case it is necessary to traverse the syntax tree several times, we can register in a linked list the sequence of stacks of symbol tables that are applicable at every point in the program.

5 Context-restricted Instructions

- Some programming languages restrict the use of certain instructions to certain contexts.
- For example, in Java the instructions `break` and `continue` can only be used inside of loops or the conditional statement `switch`.
- The semantic check of this condition is very simple to implement and can be done during the analysis syntactical using semantic predicates and a counter (or a stack) that registers the context.

```
@parser::members {
   int acceptBreak=0;
}
...
forLoop: 'for' '(' expr ';' expr ';' expr ')'
         {acceptBreak++;}
         instruction
         {acceptBreak--;}
       ;
break: {acceptBreak > 0}? 'break' ';'
     ;
instruction: forLoop | break | ...
           ;
```