Exercises on Compilers

Jordi Cortadella

February 7, 2022

Attribute grammars

---

1. Write an attribute grammar to specify binary numerals. A binary numeral is a non-empty sequence of binary digits followed by a period and another non-empty sequence of binary digits. The grammar must calculate an attribute that returns the value of the binary numeral (a real number). Use the following context-free grammar:

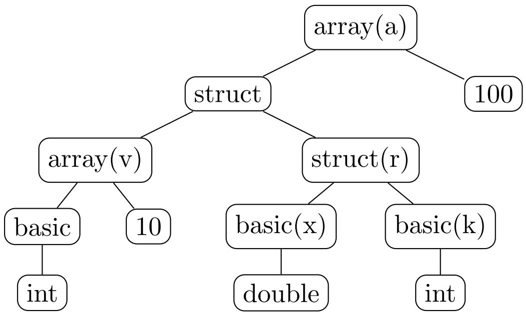
$$\begin{aligned} \texttt{Num} &\rightarrow \texttt{Digits '.' Digits} \\ \texttt{Digits} &\rightarrow \texttt{Bit} \mid \texttt{Digits Bit} \\ \texttt{Bit} &\rightarrow \texttt{'0'} \mid \texttt{'1'} \end{aligned}$$

Solution:
For every non-terminal symbol of the grammar we define the attribute `value` that will store the numerical value of the string represented by that symbol. For the symbol `Digits` we also define the attribute `length` that will represent the number of bits of the string.

$$\begin{aligned} \texttt{Num} &\rightarrow \texttt{Digits}_1\ \texttt{'.'}\ \texttt{Digits}_2 \\ & \quad \{\texttt{Num.value = Digits}_1\texttt{.value + Digits}_2\texttt{.value}/2^{\texttt{Digits}_2\texttt{.length}}\texttt{;}\} \\ \texttt{Digits} &\rightarrow \texttt{Bit} \quad \{\texttt{Digits.length= 1; Digits.value = Bit.value;}\} \\ \texttt{Digits}_1 &\rightarrow \texttt{Digits}_2\ \texttt{Bit} \\ & \quad \{\texttt{Digits}_1\texttt{.length = Digits}_2\texttt{.length + 1;}\} \\ & \quad \{\texttt{Digits}_1\texttt{.value = 2}\cdot\texttt{Digits}_2\texttt{.value + Bit.value;}\} \\ \texttt{Bit} &\rightarrow \texttt{'0'} \quad \{\texttt{Bit.value = 0;}\} \\ \texttt{Bit} &\rightarrow \texttt{'1'} \quad \{\texttt{Bit.value = 1;}\} \end{aligned}$$

2. Write an attribute grammar to construct a syntax tree for data types with arrays and structs (see example). Assume that only `int` and `double` basic types are used.

```
struct {
  int [10] v;
  struct {
    double x;
    int k;
  } r;
} [100] a;
```



Synthesize an attribute that indicates the size of the data structure, assuming that an `int` takes 4 bytes and a `double` takes 8 bytes.

Solution:
We assume there are two tokens, `ID` and `NUM`, that represent the identifiers and the positive numbers, respectively. `ID` has a string atribute `ID.name` with the value of the identifier, whereas `NUM` has an integer attribute `NUM.value` that represents the value of the number.
We first define the attribute grammar that creates the syntax tree. For that, we assume that all non-terminal symbols (except `fields`), have an attribute `node` that points at the syntax tree node. The symbol `fields` has a `list` attribute that contains a list of tree nodes.
The `node` attributes are synthesized, whereas the `list` attribute is inherited. When a production has two actions (different blocks of curly brackets), the first one is assumed to be executed before parsing the production.

```
   decl → type ID ';'              {decl.node = type.node; decl.node.name = ID.name}

   type → basic                    {data.node = basic.node}
        → struct                   {data.node = struct.node}
        → array                    {data.node = array.node}

  basic → int                      {basic.node = CreateNode('int')}
        → double                   {basic.node = CreateNode('double')}

 struct → 'struct' '{' fields '}'  {fields.list = CreateList()}
                                   {n = CreateNode('struct');}
                                   {n.AddChildren(fields.list); struct.node = n}

fields1 → decl fields2             {fields2.list = fields1.list    // copy pointer}
                                   {fields1.list.append(decl.node)}
        → decl                     {fields1.list.append(decl.node)}

  array → type '[' NUM ']'         {n = CreateNode();}
                                   {n.AddChildren(type.node, NUM.value); array.node = n}
```

The same grammar is next presented. The actions to synthesize the attribute `size` are defined in this one.

```
decl     → type ID ';'                  {decl.size = type.size}

type     → basic                        {data.size = basic.size}
         → struct                       {data.size = struct.size}
         → array                        {data.size = array.size}

basic    → int                          {basic.size = 4}
         → double                       {basic.size = 8}

struct   → 'struct' '{' fields '}'      {struct.size = fields.size}

fields1  → decl fields2                 {fields1.size = decl.size + fields2.size}
         → decl                         {fields1.size = decl.size}

array    → type '[' NUM ']'             {array.size = NUM.value * type.size}
```