Exercise 3

Consider the language of sequences of symbols on the alphabet {&, A, B, C}. The character '&' is used as a quote character to indicate that the next symbol, if different from '&', must be interpreted as a command. The sequence '&&' is used to denote the symbol '&' itself.

Reply to the following questions:

1. Give a grammar for the language
2. Give a Syntax Directed Translation Scheme, suitable for being implemented during top-down parsing, that computes as an attribute of the start symbol the sequence that is obtained resolving the commands and the quoting. For instance, for the input sequence &&&A&&B the computed attribute must be: &Cmd(A)&B where Cmd is the operator to encapsulate commands.
3. Give a Syntax Directed Translation Scheme, suitable for being implemented during bottom-up parsing, that computes the same attribute of step 2)
4. Observe that the sequences of the language should not end with a dangling '&', i.e., a '&' symbol that was not preceded by another '&' symbol. Discuss the solution of step 3) when this constraint is imposed.

Use the operator "_" for constructing strings and the operator + to concatenate two strings.

Hints

Exercise 3.1

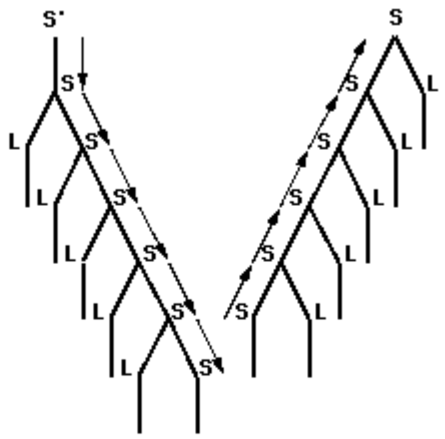
We can give a simple grammar using a linear recursion. It is better to start with a grammar that is LL(1), i.e., using a linear right recursion. This is suitable for the top-down parsing. Later, another grammar that is LR(1) will be given for the bottom-up parsing.

Esercizio3.2

The chosen grammar permits to parse only one symbol at each step through the production S_1::= L S_2. This imposes that S must record the symbol previously read, thus requiring a boolean inherited attribute *cmd* that is true iff the last symbol read was a '&'. A synthesised attribute *string* can be used to compute the resulting string using the value of *cmd*.

Exercise 3.3

The solution given in 3.2 uses an inherited attribute cmd. The figure below shows in the left part the dependency graph of this attribute for the parsing of a string of the language of length 5. Markers could be used with the same grammar by transforming the inherited attribute into a synthesised one of the markers and by expressing the actions as functions with side-effect. Another alternative solution is to change the grammar in such a way that the parse tree looks like the one on the right part of the figure. This form permits to treat the former inherited attribute as a synthesised.



Exercise 3.4

The grammar must be modified and the SDT adapted. The new grammar must permit to read the input looking at two adjacent symbols in the same step. The solution can be given as an instance of a static analysis, i.e., the defined attributes are used to control that the analysed string belongs to the described sub-language.

Solutions

Exercise 3.1

S::= L S
S::= ε
L::= &
L::= A
L::= B
L::= C

The grammar is LL(1) suitable for top-down parsing.
Another grammar, which is LR(1), i.e. suitable for bottom-up parsing, is the following:

S::= & L S
S::= T L S
S::= L
S::= ε
L::= T
L::= &
T::= A
T::= B
T::= C

This second grammar has the advantage of permitting to look at two adjacent symbols in the same step.

Exercise 3.2

We use three attributes:

- **cmd** for S: inherited, contains true if the last symbol read before the derivation of S is '&'
- **string** for S: synthesised, contains the sequence of traversed symbols resolved with respect to the quoting symbol '&'
- **val** for L: synthesised, contains the string of the traversed symbol.

We need to modify the grammar by introducing a fake start symbol S' to initialize the inherited attribute.

S'::= {S.cmd:= false}
S
{S'.string:= S.string}

S_1::= **L** {S_2.cmd:= (not S_1.cmd) and (L.val = "&")} **S_2** {S_1.string:= if (S_2.cmd) then S_2.string else if (S1.cmd) and (L.val <> "&") then "Cmd("+L.val+")"+S_2.string else L.val+S_2.string}
S::= **ε**{S.string:= if (S.cmd) then "&" else ""}
L::= **&**{L.val:= "&"}
L::= **A** {L.val:= "A"}
L::= **B** {L.val:= "B"}
L::= **C** {L.val:= "C"}

Exercise 3.3

Solution with the markers.

1. Let's start with the solution for top-down parsing given in 3.2
2. Let's insert the markers and the epsilon-productions.

The following is the solution

```
S'::= M1
      S
        {S'.string:= S.string}
---------------------------------------------------------------
S1::= L
      M2
      S2
  {S1.string:= if (M2.cmd) then S2.string
                        else if (S1.cmd) and (L.val <>
"&") then "cmd("+L.val+")"+S2.string
                              else L.val+S2.string}
---------------------------------------------------------------
S::= ε{S.string:= if (S.cmd) then "&" else ""}
---------------------------------------------------------------
L::= &{L.val:= "&"}
---------------------------------------------------------------
L::= A  {L.val:= "A"}
---------------------------------------------------------------
L::= B  {L.val:= "B"}
---------------------------------------------------------------
L::= C  {L.val:= "C"}
---------------------------------------------------------------
M1::= ε  {M1.cmd:= false}
---------------------------------------------------------------
M2::= ε  {M2.cmd:= (not S1.cmd) and (L.val = "&")}
```

3. Let’s resolve the references to the attributes through references on the stack and actions with side-effects. We suppose the following invariant: “each handle $A::=B_1,...,B_k$, has synthesises attributes of B_j in Top\j”.

```
S'::= M1
      S
        {temp:= Top; pop(2); push(temp)}
---------------------------------------------------------------
S1::= L
      M2
      S2
  {temp:= if (Top\1) then Top
                        else if (Top\3) and (Top\2 <>
"&") then "cmd("+Top\2+")"+Top
                              else Top\2+Top;
          pop(3);
          push(temp)}
---------------------------------------------------------------
S::= ε  {if (Top) then push("&") else push("")}
---------------------------------------------------------------
L::= &{push("&")}
---------------------------------------------------------------
L::= A  {push("A")}
---------------------------------------------------------------
L::= B  {push("B")}
---------------------------------------------------------------
L::= C  {push("C")}
```

M_1::= **ε** {push(false)}
M_2::= **ε** {push((not Top\1) and (Top = "&"))}

The actions pop and push maintain the invariant. This solution is completely automatic and is the solution adopted in the tools of Cornell Syntetizer

Alternative solution

Let’s write a new grammar in such a way that the inherited attribute is no longer needed.

S_1::= **S_2** **L** {S_1.cmd:= (not S_2.cmd) and (L.val = "&") S_1.string:= if ((not S_2.cmd) and (L.val = "&")) then S_2.string else if (S_2.cmd) and (L.val <> "&") then S_2.string + "cmd("+L.val+")" else S_2.string+L.val}
S::= **ε**{S.cmd:=false; S.string:= ""}
L::= **&**{L.val:= "&"}
L::= **A** {L.val:= "A"}
L::= **B** {L.val:= "B"}
L::= **C** {L.val:= "C"}

Exercise 3.4

The following grammar corresponds to the one in 3.1 but excludes strings ending with 'X&' where X can be A, B or C.

S::= T S
S::= & L S
S::= ε
L::= &
L::= T
T::= A
T::= B
T::= C

The SDT is modified as follows.

S'::= **M_1**

S {temp:= Top; pop(2); push(temp)}
S_1::=T **M_2** **S_2** {temp:= if (Top\3) then "cmd("+Top\2+")"+Top else Top\2+Top; pop(3); push(temp)}
S_1::= & **L** **M_3** **S_2** {temp:= if (Top\1) then "&"+Top else if (Top\2 <> "&") then "cmd("+Top\2+")"+Top else Top\2+Top; pop(3); push(temp)}
S::= **ε** {if (Top) then push("&") else push("")}
L::= **&**{push("&")}
L::= **T**{} ******note: we should copy T, remove it from top and push the copied value ******
T::= **A** {push("A")}
T::= **B** {push("B")}
T::= **C** {push("C")}
M_1::= **ε** {push(false)}
M_2::= **ε** {push(false)}
M_3::= **ε** {push((Top\1) and (Top = "&"))}

As an alternative solution let's extend the grammar of 3.1 using a synthesised attribute ok that tells if the string belongs to the required sub-language

S_1::= S_2 **L** {S_1.cmd:= (not S_2.cmd) and (L.val = "&") S_1.string:= if ((not S_2.cmd) and (L.val = "&")) then S_2.string else if (S_2.cmd) and (L.val <> "&") then S_2.string + "cmd("+L.val+")"

else S_2.string+L.val S_1.ok:= (L.val <> "&") or (S_2.cmd)}

S::= **ε**{S.cmd:=false;
S.string:= ""
S.ok:=true}

L::= **&**{L.val:= "&"}
L::= **A** {L.val:= "A"}
L::= **B** {L.val:= "B"}
L::= **C** {L.val:= "C"}