UNITS 9, 10 and 11: CODE OPTIMIZATION and GENERATION

Exercise: Deferred code generation.

You are asked to design an arithmetic expression calculator that translates the input sequence (infix format) into a postfix or prefix format. This type of problem is solved by carefully choosing the insertion point in each grammar rule of the semantic actions that print the translated fragments.

But this process can be very complicate due to factored productions or when the grammar grows, and especially for the prefix format.

A suggestion is to use dynamic memory to collect the translated fragments, and to rearrange them at the end of the parsing process.

David Griol Barres, Antonio Berlanga de Jesús, Jesús García Herrero, Juan Manuel Alonso Weber

SOLUTION

We propose the exercise to show how to use dynamic memory to generate code in a way that helps to simplify the placement of semantic actions. For this purpose, we choose printing the translated code to a temporary output, which allows us to rearrange the translated fragments later. The temporary output can be implemented using temporary files, or vectors of character type (string). For a more effective processing, we choose character-type vectors in dynamic memory.

To apply this solution, we will use a temporary character vector (a global variable) in which we print a code fragment (the one corresponding to a node in the syntactic tree). Here we will use the function `sprintf`, that prints on a vector, or the functions `strcpy` and `strcat`.

When the translated fragment should be transmitted to an upper node of the syntactic tree, the contents of the temporal vector must be copied into dynamic memory and return a pointer to that fragment in dynamic memory to transmit it as an attribute. For this we will use the function `gen_cad` (the same as `strdup`).

In the upper nodes, the fragments received in dynamic memory can be joined using `sprintf, strcpy` and `strcat` to create more complete and elaborate translations.

The output of the translated code will be done at the end of the input.

The advantage of this scheme is that to convert the infix to postfix converter into a translator that generates prefix or infix output, you only have to change the order in which each translated fragment is joined:

For infix output:

```
expression:   …
           | operand operator expression  { strcpy (temp, "") ;
                                            strcat (temp, $1.code) ;
                                            strcat (temp, $2.code) ;
                                            strcat (temp, $3.code) ;
                                            $$.code = gen_cad (temp) ; }
```

For postfix output:

```
expression:   …
           | operand  operator expression  { strcpy (temp, "") ;
                                             strcat (temp, $1.code) ;
                                             strcat (temp, $3.code) ;
                                             strcat (temp, $2.code) ;
                                             $$.code = gen_cad (temp) ; }
```

For prefix output:

```
expression:   …
           | operand  operator expression  { strcpy (temp, "") ;
                                             strcat (temp, $2.code) ;
                                             strcat (temp, $1.code) ;
                                             strcat (temp, $3.code) ;
                                             $$.code = gen_cad (temp) ; }
```

A disadvantage is that this method requires a much more elaborate dynamic memory management. Once a new fragment is created in dynamic memory, those fragments that correspond to its components should be disposed. But, for expressions with very deep syntactic trees, the dynamic memory consumption can increase exponentially until exhaustion when ascending in the tree.

The most logical solution would be to represent the translation in a tree structure, rather than in a linear structure. In this way, each generated fragment would be "definitive" in its corresponding node of the **abstract syntax tree**.