The Duff parser

Bart Offereins, s2255243
Marco Gunnink, s2170248

March 11, 2014

Assignment

The assignment is to make a parser for the ANSI-C language and convert the input to 'pretty' LaTeX output. The parser has to make use of the scanner we've built in lab assignment 1. The recommended tool for this is the Accent Compiler Compiler. Since it is not so easy to use the output of the gcc preprocessor, we don't have to handle prepocessor directives.

Design

To make a scanner and parser compatible, they at least have to use the same tokens. Therefore we removed the token definition enum from the scanner input and put it in a token directive in the accent file. Accent then generates a header file containing these definitions, which we included in the scanner file. The next modification we made to the scanner is to put the lexemes int to a special variable yylval. We defined the type of this yylval as a union, so it could contain all the different types of lexemes. In addition to the ususal lexeme types (eg. identifers, strings and numbers) we put the keywords, operator and other interpunction characters in the yylval so as to make printing of those more convenient. We use the grammar specification from 'The C programming language' as a basis for our parser. We rewrote it to make use of the regular expression facilities offered by Accent. Since there is a difference in the escape sequences between LaTeX and C, we had to convert certain special characters. For this we made a few helper functions. To keep the grammar readable, we defined some macros for commonly used functions to add more brevity. These are then called in the semantic actions of the parser and so produced a final LaTeX result.

Our parser accepts pretty much all valid ANSI-C code.

Result

The following code is the result of running some sample code trough our parser.

articleThe accent compiler compilerLatex pretty printer

```
int printf(char *fmt);
 void quick_sort(int arr [20], int low , int high ){
   int pivot, j, temp, i;
   if (low < high){
      pivot = low;
      i = low;
      j = high;
      while (i < j){
         while ((arr[i] ≤ arr[pivot]) ∧ (i < high)){
            i++;
         }
         while (arr[j] > arr[pivot]){
            j–;
         }
         if (i < j){
            temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
         }
         else if (derp){
            bjork();
         }
         else {
            derp();
         }
      }
      temp= arr[pivot];
      arr[pivot]= arr[j];
      arr[j]= temp;
      quick_sort(arr, low, j - 1);
      quick_sort(arr, j + 1, high);
   }
}

int main(int argc , char *argv[]){
   int i, a[10] = {1, 5, 3, 8, 6, 2, 8, 6, 4, 1};
   register n = (count + 7) / 8;
   quick_sort (a, 0, 9);
   for (i= 0; i < 10; i++){
      printf (" %4d", a[i]);
   }
   putchar('\n');
   switch (count % 8){
```

```
      case 0:do {
        to++= *from++;
        case 1:*to++= *from++;
        case 2:*to++= *from++;
        case 3:*to++= *from++;
        case 4:*to++= *from++;
        case 5:*to++= *from++;
        case 6:*to++= *from++;
        case 7:*to++= *from++;
      }
      while (–n > 0);
    }
    return 0;
}
```