Parsing and AST construction with PCCTS

Guillem Godoy and Ramon Ferrer i Cancho

February 22, 2010

These pages introduce the use of PCCTS for constructing descendant parsers and AST generators. It follows a practical approach: to learn by writing and modifying examples. PCCTS includes the parser generator program `antlr`, and the scanner or lexical analyser generator program `dlg` and much other utilities. The main web page of PCCTS is `http://www.polhode.com/pccts.html`.

1 Our first PCCTS program

Open a first file called `example0.g` and write the following PCCTS program.

```
#header
<<
#include "charptr.h"
>>

<<
#include "charptr.c"

int main() {
    ANTLR(expr(), stdin);
}
>>

#lexclass START
#token NUM "[0-9]+"
#token PLUS "\+"
#token SPACE "[\ \n]" << zzskip(); >>
```

```
expr: NUM (PLUS NUM)* ;
```

This program contains a grammar definition `expr: NUM (PLUS NUM)*` representing the expressions of sums of natural numbers. The definitions of the tokens `NUM and PLUS` also appear, and another token `SPACE` that represents the possible separators of other tokens in the input. The command `zzskip()` indicates to the scanner that the token `SPACE` does not have to be passed to the parser.

For compiling:

```
antlr example0.g
dlg parser.dlg scan.c
gcc -o example0 example0.c scan.c err.c
```

The command `antlr example0.g` generates the file `example0.c`, containing the parser itself. It also generates the file `parser.dlg`, containing the scanner definition. Moreover, it generates some additional auxiliar files `err.c` and `tokens.h`.

The command `dlg parser.dlg scan.c` generates the file `scan.c`, which contains the scanner, i.e. a function that reads the input and generates a list of tokens that will be passed to the parser. This command generates also the auxiliary file `mode.h`.

The call to `gcc` generates the executable file.

In order to understand the generated program, open the file `example0.c`. You should find a C function called `expr`, which is the natural translation of the previous `expr` variable of the grammar, and whose goal is to recognise a word in the input generable by such a variable. This function should have the following appearance, but slightly more cryptic.

```
void expr() {
    MATCHTOKEN(NUM);
    while (LOOKAHEAD() == PLUS) {
        MATCHTOKEN(PLUS);
        MATCHTOKEN(NUM);
    }
}
```

Execute the program `example0` and observe its behaviour with different inputs, like `3 + +`, for which it should exhibit a syntax error, whereas for an input like `3 + 4` it should work well. Use the `control-d` character to mark the end of the input if you use the keyboard as the standard input. Nevertheless, an input `3 3` will not produce error, even thought it is incorrect. Think on why is happening that in base to the previously generated program. In order to detect such situations, impose the end of file character also in the grammar:

```
expr: NUM (PLUS NUM)* "@" ;
```

or alternatively

```
input: expr "@" ;
expr: NUM (PLUS NUM)* ;
```

Exercise:

- Try with different grammar definitions of the same language like `expr: expr PLUS expr | NUM` or `expr: NUM PLUS expr | NUM` or `expr: expr PLUS NUM | NUM` or `expr: NUM | NUM PLUS expr`. All them generate problems. For every case, look at the generated code (that for sure will have recursive rules to `expr`) and understand why it does not work. Propose an alternative definition using recursive calls that works.
- Add the subtraction operator (binary minus operator). Compile with the `-gs` option (`antlr -gs example0.g`) in order to forbid certain optimisations that make a less comprehensive code if you want to look at it.

2 Automatic tree construction

Write the following variant of the initial PCCTS program example, where some additional definitions are given in order to deal with the `PCCTS` automatic tree construction feature. We will use some data structures of C++, and hence we will compile accordingly.

The definitions in the header area specify the type of an attribute related to a token (`Attrib`), the declaration of the function that constructs such

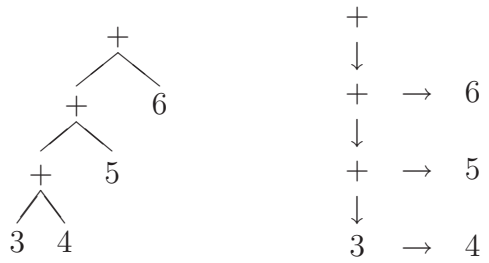

Figure 1: Abstract and internal tree representation of the expression $3+4+5+6$ assuming left-associativity.

an attribute from the input text (`zzcr_attr`), the user fields of a node of the *Abstract Syntax Tree (AST)* (`AST_FIELDS`), and the macro/function that creates such a node. There are two automatically existing fields, `right` and `down`, that define the structure of the AST and allow to manage it. For example, the expression $3 + 4 + 5 + 6$ has the following structure (left of Fig. 1), after the implicit parenthesization $((3 + 4) + 5) + 6$ assuming left-associativity. We also have its internal representation with `PCCTS` using `down` for children and `right` for brothers (right of Fig. 1).

Continuing with the `PCCTS` program, next, we have the concrete definition of `zzcr_attr`, but we have included also a function `ASTPrint` for writing the structure and contents of the AST on the standard output.

```
#header
<<
#include <string>
#include <iostream>

using namespace std;

typedef struct {
```

```
    string kind;
    string text;
} Attrib;
void zzcr_attr(Attrib *attr, int type, char *text);

#define AST_FIELDS string kind; string text;
#define zzcr_ast(as, attr, ttype, textt) as = new AST;\
(as)->kind = (attr)->kind; (as)->text = (attr)->text;\
(as)->right = NULL; (as)->down = NULL;
>>

<<
#include <cstdlib>
#include <cmath>

void zzcr_attr(Attrib *attr, int type, char *text) {
    attr->kind = text;
    attr->text = "";
    if (type == NUM) {
        attr->kind = "intconst";
        attr->text = text;
    }
}

void ASTPrintIndent(AST *a, string s) {
    if (a == NULL) return;
    cout << s << " " << a->kind;
    if (a->text != "") cout << "(" << a->text << ")";
    cout << endl;
    ASTPrintIndent(a->down, s + " |");
    ASTPrintIndent(a->right, s);
}

void ASTPrint(AST *a) {
    cout << endl;
    ASTPrintIndent(a, "");
}
```

```
int main() {
    AST *root = NULL;
    ANTLR(expr(&root), stdin);
    ASTPrint(root);
}
>>

#lexclass START
#token NUM "[0-9]+"
#token PLUS "\+"
#token SPACE "[\ \n]" << zzskip();>>

expr: NUM (PLUS NUM )* ;
```

For compiling (note that we use `g++` now instead of `gcc`):

```
antlr -gt example1.g
dlg -ci parser.dlg scan.c
g++ -o example1 example1.c scan.c err.c
```

For removing all generated files:

```
rm -f *.o example1.c scan.c err.c parser.dlg tokens.h mode.h
```

Try the generated program with several inputs. With an input as:

```
3+4+5+6
```

the resulting tree is just the list of tokens:

```
 intconst(3)
 +
 intconst(4)
 +
 intconst(5)
 +
 intconst(6)
```

Unfortunately we are not distinguishing differences between tokens, neither inferring some structure from them.

If our intention was to keep just the numbers and remove the operators, the ! operator is what we need, since it allows one to mark tokens to not be added to the sibling list.

```
expr: NUM (PLUS! NUM )* ;
```

But what we prefer is to transform the input into a conceptually clearer tree. Modify the grammar definition including the symbol ^ as follows, to indicate that a concrete token passes to be the root of the current tree, leaving the sibling list up to now as a child. The next elements will be also part of the sibling list unless a new root is indicated.

```
expr: NUM (PLUS^ NUM )* ;
```

A conceptually better AST is obtained then and printed as follows. It coincides with the internal representation with right and down we have seen before: each down adds two spaces for the tabulation of the childs.

```
+
| +
| | +
| | | intconst(3)
| | | intconst(4)
| | intconst(5)
| intconst(6)
```

Exercises:

- Include other operators and parenthesis in your grammar definition, similarly to the previous section, giving them the usual priority and left-parenthesising associativity. For example, if one has the product operator (`TIMES`), it is convenient to comply with the following structure

  ```
  expr: term (PLUS^ term)* ;
  term: NUM (TIMES^ NUM)* ;
  ```

  in order to construct a good tree according to the customary priority of the product over the sum operator. Check that the generated trees are correct.

- Try also the right-parenthesising version:

  ```
  expr: NUM (PLUS^ expr | ) ;
  ```

The expected AST for the same previous example should be:

```
+
| intconst(3)
| +
| | intconst(4)
| | +
| | | intconst(5)
| | | intconst(6)
```

Include also other operators with the usual priority between them. Note that then we are changing the meaning of the interpretation of the input if the subtraction operator (binary minus operator) appears.

Now, remove the previous `main` and add the following, in order to do an evaluation/interpretation of the AST:

```
int evaluate(AST *a) {
    if (a == NULL) return 0;
    if (a->kind == "intconst") return atoi(a->text.c_str());
    if (a->kind == "+") {
        return evaluate(a->down) + evaluate(a->down->right);
    }
}

int main() {
    AST *root = NULL;
    ANTLR(expr(&root), stdin);
    ASTPrint(root);
    cout << evaluate(root) << endl;
}
```

Exercises:

- Include the treatment of other operators as before.
- Try the right-parenthesising version, and see what happens with the binary minus operator. Do you obtain the expected results in expressions involving this operator?

As a final goal, we want to deal with programs that are sequences of instructions.

```
x:=3+5
write x
y:=3+x+5
write y
```

To this end we will need three new tokens

```
#token WRITE "write"
#token ID "[a-zA-Z]"
#token ASIG ":="
```

The order between `WRITE` and `ID` is important because of the ambiguity of having both definitions. Writing `WRITE` first we give more priority to it.

We need also to change the initial structure of the grammar:

```
program: (instruction)* ;

instruction: ID ASIG^ expr | WRITE^ expr ;
```

Note that, although the following definition is also correct, the corresponding generated abstract tree is not fine.

```
program: (ID ASIG^ expr | WRITE^ expr)* ;
```

But an identifier can also be part of the expression, and hence, change the grammar accordingly. Moreover we have to modify the function

```
void zzcr_attr(Attrib *attr, int type, char *text)
```

accordingly.

We have also to change the main function in order start the parsing from the new rule `program` and to execute the program.

```
int main() {
    AST *root = NULL;
    ANTLR(program(&root), stdin);
    ASTPrint(root);
    execute(root);
}
```

The easiest way of keeping a list of pairs ⟨`symbol`,`value`⟩ is to use the abstract data structure `map`. Hence, we need to add the following include:

```
#include <map>
```

and globally declare

```
map<string, int> m;
```

To avoid compilation problems, put the declaration in the second C code area of the .g file.

We use this simple version of symbol table in the execution:

```
void execute(AST *a) {
    if (a == NULL) return;
    if (a->kind == ":=") {
      m[a->down->text] = evaluate(a->down->right);
    }
    else { // a->kind == "write"
      cout << evaluate(a->down) << endl;
    }
    execute(a->right);
}
```

The function `evaluate` has to be modified accordingly for the cases where the expression is an identifier.

Exercise: Include other operators as before, but also other kind of instructions, like `while exp do linst endwhile` and `if exp then linst endif`. What would it happen if you included `if exp then linst` instead of `if exp then linst endif`? Try it.