Parsing and AST construction with PCCTS

Guillem Godoy and Ramon Ferrer i Cancho

September 12, 2011

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 many other utilities. The main web page of PCCTS is `http://www.polhode.com/pccts.html`.

1 Our first PCCTS program

Our first example program in PCCTS is in the file `example0.g`:

```
#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)* ;
```

The program `example0.g` 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)* ;
```

Exercises:

- Try with different grammar definitions of the same language like:
  - `expr: expr PLUS expr | NUM;`
  - `expr: NUM PLUS expr | NUM;`
  - `expr: expr PLUS NUM | NUM;`
  - `expr: NUM | NUM PLUS expr;`

  All of them generate problems. For every case, look at the generated code (that for sure will have recursive calls to `expr`) and understand why it does not work. Propose a working alternative definition using recursive calls.

- Add the subtraction operator (binary minus operator). If you want to look at the code, compile with the `-gs` option (`antlr -gs example0.g`) in order to forbid certain optimisations that produce a less readable code.

2 Automatic tree construction

Now we will use the program `example1.g` to learn how to handle syntax trees generated by PCCTS. This program is a variant of the previous example with some additional definitions needed to deal with the `PCCTS` automatic tree construction feature. We will use some C++ code, and hence we will compile accordingly.

The definitions in the `#header` area must specify:

- The declaration of a structure named `Attrib` containing the attributes related to a token.
- The declaration of a function named `zzcr_attr` that builds the `Attrib` structure for a token using the input text and the identified token code.
- A macro named `AST_FIELDS` with the definition of user-defined fields in each node of the AST (*Abstract Syntax Tree*)
- A macro named `zzcr_ast` that creates such an AST node from the token `Attrib`.

The AST nodes contain also two automatically created fields: `right` and `down`, which define the structure of the AST and allow to manage it. For example, the expression `3+4+5+6` has the structure presented in Figure 2 (left), after the implicit parenthesization `((3+4)+5)+6` assuming left-associativity. On the right-hand side of Figure 2, we can see the internal representation of the AST using `down` for the first child of each node and `right` for its next sibling.

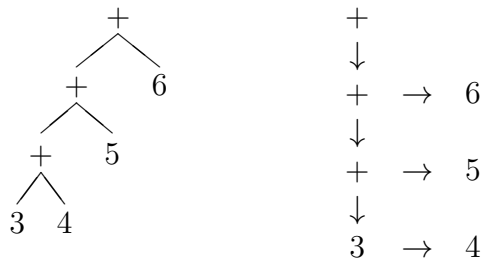

Figure 1: Abstract (left) and internal (right) tree representation of the expression `3+4+5+6`, assuming left-associativity.

With all this additions, plus some functions to output the trees in a readable format, the program `example1.g` results as follows:

```
#header
<<
#include <string>
#include <iostream>
using namespace std;

// struct to store information about tokens
typedef struct {
  string kind;
  string text;
} Attrib;

// function to fill token information (predeclaration)
void zzcr_attr(Attrib *attr, int type, char *text);

// fields for AST nodes
#define AST_FIELDS string kind; string text;
#include "ast.h"

// macro to create a new AST node (and function predeclaration)
#define zzcr_ast(as,attr,ttype,textt) as=createASTnode(attr,ttype,textt)
AST* createASTnode(Attrib* attr, int ttype, char *textt);
>>

<<
#include <cstdlib>
#include <cmath>
// function to fill token information
void zzcr_attr(Attrib *attr, int type, char *text) {
  if (type == NUM) {
    attr->kind = "intconst";
    attr->text = text;
  }
  else {
    attr->kind = text;
    attr->text = "";
  }
}

// function to create a new AST node
AST* createASTnode(Attrib* attr, int type, char* text) {
  AST* as = new AST;
  as->kind = attr->kind;
```

```
  as->text = attr->text;
  as->right = NULL; as->down = NULL;
  return as;
}

/// get nth child of a tree. Count starts at 0.
/// if no such child, returns NULL
AST* child(AST *a,int n) {
 AST *c=a->down;
 for (int i=0; c!=NULL && i<n; i++) c=c->right;
 return c;
}

/// print AST, recursively, with indentation
void ASTPrintIndent(AST *a,string s)
{
  if (a==NULL) return;

  cout<<a->kind;
  if (a->text!="") cout<<"("<<a->text<<")";
  cout<<endl;

  AST *i = a->down;
  while (i!=NULL && i->right!=NULL) {
    cout<<s+"  \\__";
    ASTPrintIndent(i,s+"  |"+string(i->kind.size()+i->text.size(),' '));
    i=i->right;
  }

  if (i!=NULL) {
      cout<<s+"  \\__";
      ASTPrintIndent(i,s+"   "+string(i->kind.size()+i->text.size(),' '));
      i=i->right;
  }
}

/// print AST
void ASTPrint(AST *a)
{
  while (a!=NULL) {
    cout<<" ";
    ASTPrintIndent(a,"");
    a=a->right;
  }
}
```

```
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)* ;
```

We will now compile with `antlr` option `-gt` (*generate tree*), and use `g++` instead of `gcc`:

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

To remove 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)
```

That is, we do not get a tree but a flat list, with no structure and no distinction between number and operator tokens.

We can control the contents and the structure of the built tree modifying the structure of the grammar, and using special operators as ! and ^.

The operator ! prevents a token from being added to the tree. For instance, with the following grammar we obtain the above flat list, but only with the numbers and without the addition operators.

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

The operator ^ enables us to control which node is going to be the root of a subtree, and produce more structured outputs. Test the grammar below, which includes the operator ^ to state that the `PLUS` tokens become 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 )* ;
```

With this grammar, a more structured and useful AST is obtained:

```
+
 \__+
 |   \__+
 |   |   \__intconst(3)
 |   |   \__intconst(4)
 |   \__intconst(5)
 \__intconst(6)
```

Exercises:

- Include the substraction operator MINUS.
- Test the right-parenthesising version, and extend it with the operator MINUS:

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

  The expected AST in this case for input sequence `3+4+5+6` is:

  ```
  +
   \__intconst(3)
   \__+
       \__intconst(4)
       \__+
           \__intconst(5)
           \__intconst(6)
  ```

Note that with this right-recursion grammar, we are changing the meaning of the interpretation of the input if the subtraction operator (binary minus operator) appears.

- Include other operators (multiplication, division) and parenthesis in your (left-parenthesising) grammar definition, 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.

3 Evaluation and interpretation of ASTs

Replace the previous `main` with the code below, in order to do an evaluation/interpretation of the AST:

```
int evaluate(AST *a) {
  if (a == NULL) return 0;
  else if (a->kind == "intconst")
    return atoi(a->text.c_str());
  else if (a->kind == "+")
    return evaluate(child(a,0)) + evaluate(child(a,1));
}

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

Exercises:

- Include the treatment of other operators as needed.
- 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 program consisting of sequences of instructions, as for example:

```
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 also need to change the structure of the grammar, adding the list of instructions:

```
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)* ;
```

Since an identifier can also be part of the expression, we need to change the grammar accordingly. Morevoer, we also have to modify the function `zzcr_attr` to appropriately handle the identifier (`ID`) tokens so that we can detect them easily in the AST.

We also have to change the `main` to start the parsing from the new `program` rule, and to call a new function `execute` to execute the program once the tree has been built:

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

Keeping variable values

The easiest way of keeping the current value for each variable is maintaining a list of pairs `<identifier,value>` accessible using the `identifier` as a key. The C++ STL `map` container is a good option to do so. Thus, we will include the appropriate header:

```
#include <map>
```

and declare a global variable in the second C++ code area of the `.g` file:

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

Then, we can use this simple symbol table in the `execute` function:

```
void execute(AST *a) {
  if (a == NULL)
    return;
  else if (a->kind == ":=")
    m[child(a,0)->text] = evaluate(child(a,1));
  else // a->kind == "write"
    cout << evaluate(child(a,0)) << endl;

  execute(a->right);
}
```

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

Exercise:

- Add other instructions, such as `while exp do linst endwhile` and `if exp then linst endif`.
- Try what happens if you define the structure `if exp then linst` instead of `if exp then linst endif`.