A Simple Parser Combinator Library in C++ (DRAFT)

Sven Eric Panitz
`www.panitz.name`

23rd December 2004

Abstract

Monadic Parser Combinators stem from functional programming. This paper exploits the ideas of parser combinators and applies them to the C++ programming language. The resulting library is extremely small, flexible and easy to use. The paper contains the complete source code of the resulting parser library. As an example a parser of N. Wirth's language PL/0 is given in terms of the parser library.

1 Introduction

Parser combinators are a technique developed for lazily evaluated functional programming languages. Some early impressive example can be found in [FL89]. A recent and very efficient implemention is the *parsec*-library for Haskell [LM01]. The idea of a parser combinator library is to construct more complex parsers by combining simpler parsers. The combination is done along the building rules of productions in a grammar. There are basically three kinds of parsers:

- atomic parsers, which accept exactly one token in the input stream.
- the sequence of two parsers.
- the alternative of two parsers.

In a production the alternative is expressed by a vertical bar `|`. For the sequence no special symbol is used.

Parser combinator libraries generally construct recursive descendant parsers with backtracking. They generally construct a list of results. Each result in this list constitutes a different pars for the grammar. An empty list denotes failure [Wad85].

Since a parser is a function, parser combinators are higher-order functions. In order to mimick the optical impression of a grammar, parser combinator libraries use overloaded operators for combinator functions.

C++ is a object-oriented language. However it is enormously flexible and allows almost every programming style. Most features known from functional languages, as operator overloading, generic types or higher order functions, are available in C++. Therefore defining a parser combinator library in C++ should be only some handcraft work. In the following section a simple such library is defined.

2 Parser Combinators in C++

In order to get a most flexible parser library we use generic types.[1] In C++ generic types are expressed via templates. The C++ compiler will instantiate type variables of a template in the source code. The instantiated source code then gets compiled. this is the so called *heterogenous* translation of generic code.

As a consequence we cannot compile general code for the parser library, but have to provide it in an header file, which needs to be included by applications.

ParsLib.h

```
#ifndef __PARSLIB_H
#define __PARSLIB_H

#include <vector>

using namespace std;
namespace name {namespace panitz{ namespace parser{
```

Within the library we define a hierarchical namespace in Java package style.

2.1 The Type of A parser

2.1.1 Constant applicative forms

Before we start to define the types of a parser, we define an auxiliary class. This will be necessary, when defining the parser of a recursive (or mutual recursive) production(s). The auxiliary class is used to express a memorizing constant function, a so called *constant applicative form.* In terms of the standard template library this is a so called *generator.* The class is generic over the result type of the function. It contains a field for the function pointer and a field for the result of the function. Two constructors are provided:

ParsLib.h

```
template <typename a>
class CAF {
  private: a(*f)();
  public:
    a result;
    a operator()(){
      if (result==NULL) result=f();
      return result;
    }
    CAF(a x):result(x){}
    CAF(a(*f)()){this->f=f;result=NULL;}
};
```

Since the capability of type inference in C++ is limited and not applied for constructors, we define a function, which wraps the constructor call for `CAF`:

---

[1] In functional programming the term *polymorphic type* is used for *generic types.* However in object oriented programming *polymorphism* is used for something else.

ParsLib.h

```
template <typename a>
CAF<a> caf(a(*f)()){return CAF<a>(f);}
```

2.1.2 Results of a parser

A parser consumes some token of the input stream and produces some pars result. The result can be of any type, but generally it will be some abstract syntax tree. The token type can be of any type. Even the stream type of the input stream could be left most general, as long it has the typical*iterator* interface of retrieving the next token and a check for the end of the stream. However thoughout htis paper we will stick to the standard class `vector` from the *standard template library*.

A pars may fail. Then it usually consumes no token from the input stream. Unlike in functional implementations we will not express failure by an empty list, but have a flag for failure in the class `ParsResult`.

We keep the class for a pars results generic over two types: the token type and the type of the actual result. The class has three fields. One contains the further still to consume token, one the actual result and one the flag for failure:

ParsLib.h

```
template <typename a,typename b>
class ParsResult{
  public:
    a result;
    vector<b> furtherToken;
    bool failed;

    ParsResult(vector<b>& furtherToken)
     :furtherToken(furtherToken),failed(true){};
    ParsResult(a result,vector<b>& furtherToken)
     :result(result),furtherToken(furtherToken),failed(false){}
};
```

We provide a function which wraps the constructor for the result of a failed pars.

ParsLib.h

```
template <typename a,typename b>
ParsResult<a,b>* fail(vector<b>& furtherToken){
  return new ParsResult<a,b>(furtherToken);
}
```

This enables us to rely on C++ type inference mechanism when constructing a failure object.

2.1.3 The general parser interface

The parser interface can be kept quite simple. It just contains one method. The method `parse` takes a vector of a generic token type and returns some `ParsResult`:

`ParsLib.h`

```
template <typename a,typename b>
class Parser{
  public: virtual ParsResult<a,b>* parse(vector<b>& xs)=0;
};
```

Different from libraries in functional programming just one result is returned. Not a list of results. If there is not a unique derivation in a grammar, then our implementation will just give one and not all of them. This is for effecency reasons. Unlike in lazily evaluated functional languages without further effort, always all derivations would be calculated. This is not desirable. However most grammars used in compiler construction have unique derivations.

In the following subsections the three basic kinds of parsers are defined.

2.2 Atomic Parsers

The most simple parser is a parser, which accepts exactly one token. This parser needs to be handcoded. This is done by way of the class `GetToken`. The class has two fields. An example of the token which is to be accepted and a equality function for this token type. The class is a subclass of `Parser` and as well a subclass of `CAF`. The result of the `CAF`-object is the object itself. This enables us to use `GetToken`-objects directly as parsers but as functions returning parsers as well.

`ParsLib.h`

```
template <typename a>
class GetToken: public Parser<a,a>,public CAF<Parser<a,a>*>{
  private:
    a token;
    bool(*eq)(a,a);
  public:
    GetToken(a token,bool(*eq)(a,a))
     :CAF<Parser<a,a>*>(this),token(token),eq(eq){}
```

The implementation of the method `parse` is straightforward. Compare (if available) the next token with the token in questaion. Construct a successful or failure parser result object depending on the comparision.

`ParsLib.h`

```
    ParsResult<a,a>* parse(vector<a>& xs){
      if (!xs.empty() && eq(token,xs[0])){
        a tok = xs[0];
        vector<a> restToken = vector<a>(xs.begin()+1,xs.end());
        return new ParsResult<a,a>(tok,restToken);
      }
      return fail<a,a>(xs);
    }
};
```

To ease use we again provide a function which wraps the constructor call:

ParsLib.h

```
template <typename a>
CAF<Parser<a,a>*> getToken(a token,bool(*eq)(a,a)){
  return new GetToken<a>(token,eq);}
```

2.3 Sequence Operator

We can define classes for parser combination. The class `Seq` is use to combine to parser as a sequence. The sequence of to parsers denotes the following: first apply the first parser. In case of Success apply the second parser to the remaining token. in case of success combine the two partial results into a common result.

The most natural result for the sequence of two parser is a pair containing the two partial results. For this purpose we can use the class `pair` from the standrad template library. Thus we get the following class header:

ParsLib.h

```
template <typename a,typename b,typename c>
class Seq
     :public Parser<pair<a,b>*,c >
     ,public CAF<Parser<pair<a,b>*,c>*>{
```

As with class `GetToken`, we do not only extend the class parser, but also the class `CAF`, with the parser as its result type.

We will not directly combine two parser objects, but two `CAF` objects, which have a parser as result. We provide two internal fields for these objects. The fileds are initialized within the constructor:

ParsLib.h

```
  private:
     CAF<Parser<a,c>*> p1;
     CAF<Parser<b,c>*> p2;

  public:
    Seq(CAF<Parser<a,c>*> p1
       ,CAF<Parser<b,c>*> p2):
       CAF<Parser<pair<a,b>*,c>*>(this)
        ,p1(p1),p2(p2){}
```

Eventually the actual method `parse` needs to be defined. Again the implementation is straightforward. The only thing to take care of is storage management. Temporary results of the two combined parses need to be deleted.

ParsLib.h

```
    virtual ParsResult<pair<a,b>*,c>* parse(vector<c>& xs){
     ParsResult<a,c>* res1 = p1()->parse(xs);
     if (!res1->failed) {
        vector<c> further = res1->furtherToken;
        a r1 = res1->result;
```

```

        ParsResult<b,c>* res2 = p2()->parse(further);
        if (!res2->failed){
         b r2 = res2->result;
         vector<c> further = res2->furtherToken;
         delete res1;delete res2;
         return
         new ParsResult<pair<a,b>*,c>(new pair<a,b>(r1,r2),further);
        }
        delete res2;
      }

      delete res1;
      return fail<pair<a,b>*,c>(xs);
    }
};
```

As we are allready used to, a function which wraps the constructor is defined. This is done by way of overloading the comma operator.

`ParsLib.h`

```
template <typename a,typename b,typename c>
CAF<Parser<pair<a,b>*,c>*> operator,
                (CAF<Parser<a,c>*>p1,CAF<Parser<b,c>*> p2){
  return ((Parser<pair<a,b>*,c>*)new Seq<a,b,c>(p1,p2));
}
```

2.4 Alternative Operator

In the same way as for the sequence combinator, we can define a class for the alternative combinator. However, we need to think of the result. In a combination of parsers, the two parser may have different result types. The result type of the combination is then either the result type of the first or of the other parser. In Haskell this can be easily expressed by the algebraic type `Either`. In C++ unfortunatly there is no corresponding standard class available. However it can be easily defined by way of C's *union* construct.

`ParsLib.h`

```
template <typename A,typename B>
class Either{
 public:
   bool isLeft;
   union LeftOrRight{A left;B right;} value;
   Either(bool isLeft):isLeft(isLeft){};
};
```

We provide two simple functions to serve as constructors for this class. A function two construct an object for the left type:

ParsLib.h

```
template <typename A,typename B>
Either<A,B>* left(A v){
  Either<A,B>* result=new Either<A,B>(true);
  result->value.left=v;
}
```

And the corresponding function for the right type:

ParsLib.h

```
template <typename A,typename B>
Either<A,B>* right(B v){
  Either<A,B>* result=new Either<A,B>(false);
  result->value.right=v;
}
```

Now we are well prepared to define the alternative combinator. Its header can analogously defined to the sequence combinator:

ParsLib.h

```
template <typename a,typename b,typename c>
class Alt:public Parser<Either<a,b>*,c>
         ,public CAF<Parser<Either<a,b>*,c>*>{
  private:
   CAF<Parser<a,c>*> p1;
   CAF<Parser<b,c>*> p2;
  public:
   Alt(CAF<Parser<a,c>*> p1, CAF<Parser<b,c>*> p2):
      CAF<Parser<Either<a,b>*,c>*>(this),p1(p1),p2(p2){}
```

The implementation is again straighforward. First apply the first parser. In case of success construct a left success result. Otherwise apply the second parser to the original input stream. This is where backtracking is done. Once more we carefully need to delete intermediate results:

ParsLib.h

```
   virtual ParsResult<Either<a,b>*,c>* parse(vector<c>& xs){
     ParsResult<a,c>* res1 = p1()->parse(xs);
     if (!res1->failed) {
       vector<c> further = res1->furtherToken;
       a r1 = res1->result;
       delete res1;
       return new ParsResult<Either<a,b>*,c>(left<a,b>(r1),further);
     }
     delete res1;
     ParsResult<b,c>* res2 = p2()->parse(xs);
     if (!res2->failed) {
      b r2 = res2->result;
      vector<c> further = res2->furtherToken;
      delete res2;
```

```
      return new ParsResult<Either<a,b>*,c>(right<a,b>(r2),further);
     }
     delete res2;
     return fail<Either<a,b>*,c>(xs);
   }
};
```

The wrapper for a constructor is again defined as an overloaded operator. The most natural choice for this operator is the vertical bar, which is used in the production rules of a grammer as well.

ParsLib.h

```
template <typename a,typename b,typename c>
CAF<Parser<Either<a,b>*,c>*>
  operator|(CAF<Parser<a,c>*>p1,CAF<Parser<b,c>*> p2){
  return ((Parser<Either<a,b>*,c>*)new Alt<a,b,c>(p1,p2));
}
```

2.5 Calculating the Result

Up to now we can express the rules of a grammar nicely. However, we can only construct parsers, which have pairs or `Either` objects as results. Generally we will want to construct some special result for certain production rules. In parser generators as yacc[Joh75] the grammar gets annotated by code, which will construct some result during the parses. In parser combinator libraries this code is attached to the production of a grammar by a further combinator. This combinator allow to express that for an successful pars a function will be applied to the result. This turns out to be a map.

We provide a further class in our library. It contains fields for a parser and a function:

ParsLib.h

```
template <typename a,typename b,typename c>
class Map:public Parser<b,c>,public CAF<Parser<b,c>*>{
  private:
    Parser<a,c>* p;
    b(*f)(a);

  public:
    Map(b(*f)(a),Parser<a,c>* p)
      :CAF<Parser<b,c>*>(this),f(f),p(p){}
```

The implementation of the actual method `parse` applies the inner parser to the input, and in case of success takes its actual result and applies the function to this, in order to constrcut the overall result. Again, intermediate results need to be deleted.

ParsLib.h

```
    virtual ParsResult<b,c>* parse(vector<c>& xs){
      ParsResult<a,c>* res1 = p->parse(xs);

      if (!res1->failed){
```

```
        a r1 = res1->result;
        vector<c> further = res1->furtherToken;
        delete res1;
        return new ParsResult<b,c>((*f)(r1),further);
      }
      delete res1;
      return fail<b,c>(xs);
    }
};
```

An operator overloading is defined for construction of `Map` objects. We decided for the operator `<<`. It is used to attach some code to a rule of the grammar.

`ParsLib.h`

```
template <typename a,typename b,typename c>
CAF<Parser<b,c>*> operator<<(CAF<Parser<a,c>*> p,  b(*f)(a)){
  return ((Parser<b,c>*)new Map<a,b,c>(f,p()));
}
```

With the new parser combinator `Map` we can express one further operator. For the alternativ combination of two parsers with the same result type, it is unnecessary to differentaite the two results through an `Either` object. We can provide a common function to extract the data stored in an `Either` object.

`ParsLib.h`

```
template <typename a>
a getLeftRight(Either<a,a>* either){return either->value.left;}
```

Now we provide an alternative combinator for two parsers with the same result type. We decided for the double vertical bar as this operator:

`ParsLib.h`

```
template <typename a,typename c>
CAF<Parser<a,c>*>
  operator||(CAF<Parser<a,c>*> p1,CAF<Parser<a,c>*> p2){
  return (p1|p2)<< getLeftRight<a>;
}
```

Finally we provide a class for the empty word production. It does not consume any token and always succeeds:

`ParsLib.h`

```
template <typename a,typename b>
class Result:public Parser<a,b>,public CAF<Parser<a,b>*>{
  public:
   a x;
   Result(a x):CAF<Parser<a,b>*>(this),x(x){}
   ParsResult<a,b>* parse(vector<b>& xs){
     return new ParsResult<a,b>(x,xs);}
```

```
};
}}}//namespace
#endif

```

That's it. We defined a complete parser library for construction of recursive descendant parsers. As a consequence grammar transscribed with our library to a parser may not contain left recursive productions.

3 Example

We give an example of how to use the parser library. As a language we will use Wirth's PL/0[Wir76]. The complete grammar is given in figure 1. It is not left recursive, such that we can directly transcribe it. However it is not left unique. There are different alternatives of a rule with common prefixes. This may lead to serious efficenciy problems.

In this example implementation we will not construct any reasonable result. A simple boolean value serves as result. Therefore we provide a generic unary function, which maps its argument to the value `true`. Furthermore an epsilon parser, which results `true` is given.

Our implementation does not differentiate between parser and lexer. The lexer is completely expressed within the parser. Productions for tokens are implemented in the parser. In order to deal with arbitrary whitespace, even a special prduction for consuming whitespace is provided. All this auxiliary code, together with some type synonous can be found in figure 2.

The tokenizer part of our parser is quite simple. Sequences of certain characters are consumed. The nested pair object is then simply mapped to the value `true`.[2] The call of the generic function `mkTrue` needs to be annotated with the concrete type instance, since the C++ type inference algorith is too weak to derive this type. the complete tokenizer part of the parser is given in figure 3.

Eventually we can define the parser for PL/0. We need to take care of recursive rules. For these a function returning the corresponding parser needs to be defined. The actual parser is then a genetor for this function. Having taken care of this, the grammar can directly be expressed in C++ code. The full implementation is given in figure 4.

We provide some `main` function for our parser.

ParsePL0.cpp

```
int main(int argc,char** argv ){
  FILE *fp;
  fp = fopen(argv[1], "r");
  vector<char> xs;

  int c = getc(fp);
  while (c != EOF) {
    xs.push_back((char)c);
    c = getc(fp);
```

---

[2] We neglected that the intermediate pair objects need to be deleted from the storage. This would have needed special versions of `mkTrue` and will blow up the code. There was not enough room for this within the paper.

$$
\begin{align}
\mathit{program} &\rightarrow \mathit{block}\ \texttt{.} \tag{1}\\
\mathit{block} &\rightarrow \mathit{constDecl}\ \mathit{varDecl}\ \mathit{procDecls}\ \mathit{statement} \tag{2}\\
\mathit{constDecl} &\rightarrow \texttt{CONST}\ \mathit{constAssignmentList}\ \texttt{;}\ \mid \epsilon \tag{3}\\
\mathit{constAssignmentList} &\rightarrow \texttt{ident}\ \texttt{=}\ \texttt{number} \tag{4}\\
&\quad \mid \texttt{ident}\ \texttt{=}\ \texttt{number}\ \texttt{,}\ \mathit{constAssignmentList} \notag\\
\mathit{varDecl} &\rightarrow \texttt{VAR}\ \mathit{identList}\ \texttt{;}\ \mid \epsilon \tag{5}\\
\mathit{identList} &\rightarrow \texttt{ident}\ \texttt{,}\ \mathit{identList} \mid \texttt{ident} \tag{6}\\
\mathit{procDecls} &\rightarrow \mathit{procDecl}\ \mathit{procDecls}\ \mid \epsilon \tag{7}\\
\mathit{procDecl} &\rightarrow \texttt{PROCEDURE}\ \texttt{ident}\ \texttt{;}\ \mathit{block}\ \texttt{;} \tag{8}\\
\mathit{statement} &\rightarrow \mathit{blockSt} \mid \mathit{callSt} \mid \mathit{ifSt} \mid \mathit{whileSt} \mid \mathit{assignSt} \mid \epsilon \tag{9}\\
\mathit{assignSt} &\rightarrow \texttt{ident}\ \texttt{:=}\ \mathit{expression} \tag{10}\\
\mathit{callSt} &\rightarrow \texttt{CALL}\ \texttt{ident} \tag{11}\\
\mathit{ifSt} &\rightarrow \texttt{IF}\ \mathit{condition}\ \texttt{THEN}\ \mathit{statement} \tag{12}\\
\mathit{whileSt} &\rightarrow \texttt{WHILE}\ \mathit{condition}\ \texttt{DO}\ \mathit{statement} \tag{13}\\
\mathit{blockSt} &\rightarrow \texttt{BEGIN}\ \mathit{statementList}\ \texttt{END} \tag{14}\\
\mathit{statementList} &\rightarrow \mathit{statement}\ \texttt{;}\ \mathit{statementList} \mid \mathit{statement} \tag{15}\\
\mathit{condition} &\rightarrow \texttt{ODD}\ \mathit{expression} \mid \mathit{expression}\ \mathit{compOp}\ \mathit{expression} \tag{16}\\
\mathit{compOp} &\rightarrow \texttt{=} \mid \texttt{<>} \mid \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=} \tag{17}\\
\mathit{expression} &\rightarrow \mathit{term}\ \mathit{expression2} \tag{18}\\
&\quad \mid \mathit{addOp}\ \mathit{term}\ \mathit{expression2} \notag\\
\mathit{expression2} &\rightarrow \mathit{addOp}\ \mathit{term}\ \mathit{expression2}\ \mid \epsilon \tag{19}\\
\mathit{addOp} &\rightarrow \texttt{+} \mid \texttt{-} \tag{20}\\
\mathit{term} &\rightarrow \mathit{factor}\ \mathit{term2} \tag{21}\\
\mathit{term2} &\rightarrow \mathit{multOp}\ \mathit{factor}\ \mathit{term2}\ \mid \epsilon \tag{22}\\
\mathit{multOp} &\rightarrow \texttt{*} \mid \texttt{/} \tag{23}\\
\mathit{factor} &\rightarrow \texttt{ident} \mid \texttt{number} \mid \texttt{(}\ \mathit{expression}\ \texttt{)} \tag{24}
\end{align}
$$

Figure 1: Grammar of PL/0.

```
  }

  cout<<pl0::program()->parse(xs)->failed<<endl;
  cout<<pl0::program()->parse(xs)->result<<endl;
  cout<<pl0::program()->parse(xs)->furtherToken[0]<<endl;
}

```

The following simple program can be used as input for the parser:

Test1.pl0

```
CONST M = 7, N = 85;
VAR I,X,Y,Z,Q,R;
```

ParsePL0.cpp

```
#include "../ParsLib.h"
#include <iostream>

using namespace name::panitz::parser ;

namespace pl0{

template <typename a>
bool mkTrue(a x){return true;}

typedef Parser<bool,char>* CP;
typedef CAF<CP> P;

typedef pair<bool,bool>* pb2;
typedef pair<pair<bool,bool>*,bool>* pb3;
typedef pair<pair<pair<bool,bool>*,bool>*,bool>* pb4;
typedef pair<pair<pair<pair<bool,bool>*,bool>*,bool>*,bool>* pb5;

bool charEq(char c1,char c2){
return c1==c2;}

P epsilon = new Result<bool,char>(true);

P gC(char c){ return getToken(c,charEq) << mkTrue<char>;}

P whiteChar = gC(’ ’)||gC(’\n’)||gC(’\t’);

CP getWhiteSpace();
P whiteSpace = caf(getWhiteSpace);
CP getWhiteSpace(){return ((whiteChar,whiteSpace)<<mkTrue<pb2> ||epsilon)() ;}

P gwC(char c){return (whiteSpace,gC(c)) << mkTrue<pb2>;}
```

Figure 2: Auxilliary definitions for PL/0 parser

```

PROCEDURE MULTIPLY; VAR A,B;
BEGIN   A := X; B := Y; Z := 0;
  WHILE B > 0 DO BEGIN IF ODD B THEN Z := Z+A; A := 2*A; B := B/2; END
END;

PROCEDURE DIVIDE;
VAR W;
BEGIN   R := X; Q := 0; W := Y;
  WHILE W = R DO W := 2*W;
  WHILE W > Y DO
   BEGIN Q := 2*Q; W := W/2; IF W = R THEN BEGIN R := R-W; Q := Q+1 END END
END;

PROCEDURE GCD; VAR F,G;
BEGIN   F := X; G := Y;
  WHILE F <> G DO BEGIN IF F<G THEN G := G-F; IF G<F THEN F := F-G; END;
  Z := F
END;

BEGIN   I:=2000;
  WHILE I<>0 DO
    BEGIN X := M; Y := N; CALL MULTIPLY;X := 25; Y := 3; CALL DIVIDE;
```

ParsePL0.cpp

```
P ifT    = (gwC(’I’),gC(’F’))                        << mkTrue<pb2>;
P thenT  = (gwC(’T’),gC(’H’),gC(’E’),gC(’N’))        << mkTrue<pb4>;
P callT  = (gwC(’C’),gC(’A’),gC(’L’),gC(’L’))        << mkTrue<pb4>;
P whileT = (gwC(’W’),gC(’H’),gC(’I’),gC(’L’),gC(’E’))<< mkTrue<pb5>;
P constT = (gwC(’C’),gC(’O’),gC(’N’),gC(’S’),gC(’T’))<< mkTrue<pb5>;
P beginT = (gwC(’B’),gC(’E’),gC(’G’),gC(’I’),gC(’N’))<< mkTrue<pb5>;
P oddT   = (gwC(’O’),gC(’D’),gC(’D’))                << mkTrue<pb3>;
P endT   = (gwC(’E’),gC(’N’),gC(’D’))                << mkTrue<pb3>;
P varT   = (gwC(’V’),gC(’A’),gC(’R’))                << mkTrue<pb3>;
P doT    = (gwC(’D’),gC(’O’))                        << mkTrue<pb2>;

P procedureT = (gwC(’P’),gC(’R’),gC(’O’),gC(’C’),gC(’E’)
               ,gC(’D’),gC(’U’),gC(’R’),gC(’E’))
   << mkTrue<pair<pair<pair<pair<pair<pair<pair<pair<bool,bool>*
      ,bool>*,bool>*,bool>*,bool>*,bool>*,bool>*,bool>*>;

P addT   = gwC(’+’);
P subT   = gwC(’-’);
P mulT   = gwC(’*’);
P divT   = gwC(’/’);
P eqT    = gwC(’=’);
P gtT    = gwC(’>’);
P ltT    = gwC(’<’);
P neqT   = (gwC(’<’),gC(’>’))                       <<mkTrue<pb2>;
P geT    = (gwC(’>’),gC(’=’))                       <<mkTrue<pb2>;
P leT    = (gwC(’<’),gC(’=’))                       <<mkTrue<pb2>;
P dotT   = gwC(’.’);
P commaT = gwC(’,’);
P semicolonT   = gwC(’;’);
P lparT  = gwC(’(’);
P rparT  = gwC(’)’);
P assignT= (gwC(’:’),gC(’=’))                       <<mkTrue<pb2>;

P alphaT = gC(’A’)||gC(’B’)||gC(’C’)||gC(’D’)||gC(’E’)||gC(’F’)||gC(’G’)
         ||gC(’H’)||gC(’I’)||gC(’J’)||gC(’K’)||gC(’L’)||gC(’M’)||gC(’N’)
         ||gC(’O’)||gC(’P’)||gC(’Q’)||gC(’R’)||gC(’S’)||gC(’T’)||gC(’U’)
         ||gC(’V’)||gC(’W’)||gC(’X’)||gC(’Y’)||gC(’Z’);

P digitT = gC(’0’)||gC(’1’)||gC(’2’)||gC(’3’)||gC(’4’)||gC(’5’)||gC(’6’)
         ||gC(’7’)||gC(’8’)||gC(’9’);


CP getNumber();
P numberT=caf(getNumber);
CP getNumber(){return ((digitT,numberT)<<mkTrue<pb2>||digitT)();}
P wnumberT=(whiteSpace,numberT)        << mkTrue<pb2>;

CP getIdent();
P identT=caf(getIdent);
CP getIdent(){return ((alphaT,identT)<<mkTrue<pb2>||alphaT)();}
P widentT=(whiteSpace,identT)        << mkTrue<pb2>;
```

Figure 3: Tokenizer part of PL/0 parser

```
          X := 84; Y := 36; CALL GCD; I:=I-1;
    END;
END.//end of everything
```

4 Conclusion

We have implemented a very simple parser combinator library in C++. The implementation could be made straighforward. This is not surprising, since C++ is flexible enough to allow many very different styles of programming. It has been once more shown that operator overloading and generic types are key features for implementation of flexible libraries. This has been pointed out several times e.g. very impressively in a talk by Guy L. Steele[Ste99].

The solution chosen for recursive productions is a bit unsatisfactory. A function needs to be defined which results the parser. The function then needs to be wrapped in an `CAF` object.

It is not very surprising that due to missing support of full type inference, and through explicit memory management the implementation is more complex than a corresponding Haskell implementation.

A more than prototypic implementation of a parser library will certainly provide more ways to express parsers, as e.g. repetitions or seperated lists.

Systematic performance tests have not been made for the library.

4.1 Related Work

A fully implementation of a parser combinator library in C++ does not seem to be available. The FC++ library[MS01] does not contain parser combinators. On the website of its descendants (*Boost.FC++*) it is noted as future work[3].

A much more ambitious work has begun by Claessen [Cla]. His implementation of a combinator parser library is done in plain C.

References

[Cla] Coen Claessen. Parser combinators in c. `http://www.cs.chalmers.se/~koen/ParserComboC/parser-combo-c.html`.

[FL89] R. Frost and J. Launchbury. Constructing natural language interpreters in a lazy functional language. *The Computer Journal*, 32(2):108–121, 1989.

[Joh75] Stephen C. Johnson. Yacc: Yet another compiler compiler. Technical Report 39, Bell Laboratories, 1975.

[LM01] Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001.

[MS01] Brian McNamara and Yannis Smaragdakis. Functional programming in c++ using the fc++ library. *SIGPLAN Notices*, 36(4):25–30, 2001.

[Ste99] Guy L. Steele. Growing a language. *Higher-Order and Symbolic Computation*, 12(3):221–236, October 1999.

---

[3] As stated on the website `http://www.cc.gatech.edu/ yannis/fc++/boostpaper/fcpp.sectlimitations.html` in december 2004

[Wad85] Phil Wadler. How to replace failure by a list of successes. In *Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 113–128. Springer, 1985.

[Wir76] Niklaus Wirth. *Algorithms + Datastructures = Programs.* Prentice-Hall, 1976.

ParsePL0.cpp

```
CP getExpression();                 P expression=caf(getExpression);
CP getTerm2();                      P term2 = caf(getTerm2);
CP getExpression2();                P expression2 = caf(getExpression2);
CP getStatement();                  P statement = caf(getStatement);
CP getStatementList();              P statementList = caf(getStatementList);
CP getBlock();                      P block=caf(getBlock);
CP getProcDecls();                  P procDecls=caf(getProcDecls);
CP getIdentList();                  P identList=caf(getIdentList);
CP getConstAssignList();            P constAssignList=caf(getConstAssignList);

P addOp  = addT || subT;
P mulOp  = mulT || divT;
P compOp = eqT  || neqT || geT  || leT  || gtT || ltT  ;

P factor = widentT || wnumberT || (lparT,expression,rparT)<<mkTrue<pb3>;

CP getTerm2(){return ((mulOp,factor,term2)<<mkTrue<pb3> || epsilon)();}

P term = (factor,term2) << mkTrue<pb2>;

CP getExpression2(){return ((addOp,term,expression2)<<mkTrue<pb3> || epsilon)();}

CP getExpression(){
 return( (term,expression2)                   <<mkTrue<pb2>
       ||(addOp,term,expression2)             <<mkTrue<pb3>)();}

P condition = (oddT,expression)               <<mkTrue<pb2>
            ||(expression,compOp,expression)  <<mkTrue<pb3>;

CP getStatementList(){
 return ( (statement,semicolonT,statementList)<< mkTrue<pb3>
        ||statement              )();}

P whileSt = (whileT,condition,doT,statement)  << mkTrue<pb4>;
P ifSt    = (ifT,condition,thenT,statement)   << mkTrue<pb4>;
P callSt  = (callT,widentT)                   << mkTrue<pb2>;
P assignSt= (widentT,assignT,expression)      << mkTrue<pb3>;
P blockSt = (beginT,statementList,endT)       << mkTrue<pb3>;

CP getStatement(){return (blockSt||callSt||ifSt||whileSt||assignSt||epsilon)();}

P procDecl = (procedureT,widentT,semicolonT,block,semicolonT) <<mkTrue<pb5>;

CP getProcDecls(){return ((procDecl,procDecls)<<mkTrue<pb2>||epsilon)();}

CP getIdentList(){return ((widentT,commaT,identList)<<mkTrue<pb3>||widentT)();}

P varDecl = (varT,identList,semicolonT)<<mkTrue<pb3>||epsilon;

CP getConstAssignList(){
 return ((widentT,eqT,wnumberT,commaT,constAssignList)<<mkTrue<pb5>
       ||(widentT,eqT,wnumberT)                       <<mkTrue<pb3>)();}

P constDecl = (constT,constAssignList,semicolonT)<<mkTrue<pb3> ||epsilon;

CP getBlock(){return ((constDecl,varDecl,procDecls,statement)<<mkTrue<pb4>)();}

P program = (block,dotT)<<mkTrue<pb2>;
}
```

Figure 4: PL/0 parser