# `C5_printf` and `C5_scanf` in C5 version 0.98.

Juan José Cabezas
PEDECIBA Informática
Instituto de Computación, Universidad de la República,
Casilla de Correo 16120, Montevideo, Uruguay
Email: jcabezas@fing.edu.uy

**Abstract**

In this paper we present the version 0.98 (September 2006) of the C5 compiler including the functions `C5_printf` and `C5_scanf`.

C5 is a superset of the C programming language. The main difference between C and C5 is that the type system of C5 supports the definition of types of dependent pairs, i.e., the type of the second member of the pair depends on the value of the first member (which is a type).

Another C5 extension is the type initialization expression which is a list of dependent pairs that can be attached to type expressions in a type declaration.

These extensions provide C5 with dynamic type inspection at run time and attribute type definition. The result is a powerful framework for generic programming.

The paper presents the improvements of the version 0.98 including the functions `C5_scanf`, `C5_printf`, `C5_lenSearch` and `C5_idxSearch`.

Keywords: *dynamic type; dependent pair type ; generic programming ; parser*

# 1 Introduction

## 1.1 Inodoro's dream

27 years ago, Inodoro Pereira –our C programming teacher– had a dream.

He was presenting the C functions `printf` and `scanf` in the undergraduate course of programming at the *Instituto de Computación* of Montevideo when his famous dream came up:

> I think that `printf` and `scanf` are poor implementations of a good idea. Their arguments are limited to atomic types without type check. In other words, the functions are unsafe and limited to a few types.
>
> I would like to see new versions of these functions with type checking and defined for the entire C type system.
>
> Let us see an example of my dream functions:

```
typedef struct NODE{
                 int element;
                 struct NODE *nect;
                 } *MyType;
  main(){
        MyType ils;
        scanf(" %MyType ", ils);
        printf(" %MyType ", ils);
        }
```

> The type `MyType` is a recursive structure used to implement a linked list in the C language. My dream `scanf` reads the standard input and constructs a linked list assigned to `ils`, provided that the input match with the values required by the type `MyType`.
>
> My dream `printf` prints the integers of the linked list in the standard output. As you can see the program does not do too much; it just prints the input.
>
> Let us suppose that there exist more dream functions:
>
> - `element_occurrences`
>   The function returns the number of occurrences of an element identified by its type name.
> - `search_element`
>   The function returns the value of the $i^{th}$ occurrence of an element identified by its type name.
> - `search_type`
>   The function returns the format string of the $i^{th}$ occurrence of an element identified by its type name.
>
> So, we can now write a more interesting dream program:

```
typedef struct NODE{
                  int element;
                  struct NODE *nect;
                  } *MyType;
    main(){
          MyType ils;
          int i;
          scanf(" %MyType ", ils);
          for(i=occurrences(" %MyType ", ils,"element")-1;i>=0;i--)
               printf(search_type(i," %MyType ", ils,"element"),
                      search_element(i," %MyType ", ils,"element"));
          }
```

This program prints the input in reverse order.

The point here is that the program above is not dependent of the type definition of `MyType`.

For instance, we can define `MyType` as a four element array of integer:

```
typedef int element;
typedef element MyType[4];
    main(){
          MyType ils;
          int i;
          scanf(" %MyType ", ils);
          for(i=occurrences(" %MyType ", ils,"element")-1;i>=0;i--)
               printf(search_type(i," %MyType ", ils,"element"),
                      search_element(i," %MyType ", ils,"element"));
          }
```

In this case, the (same) program reads 4 integers and prints them in reverse order.

Further more, we can change the type of the elements and the program still works:

```
typedef char * element;  /* string */
typedef element MyType[4];
    main(){
          MyType ils;
          int i;
          scanf(" %MyType ", ils);
          for(i=occurrences(" %MyType ", ils,"element")-1;i>=0;i--)
               printf(search_type(i," %MyType ", ils,"element"),
                      search_element(i," %MyType ", ils,"element"));
          }
```

This program reads the input `one two three four` and prints –as we expect– `four three two one` .

> I don't know if my dream functions are implementable in C.
> However, this is my dream.

Inodoro Pereira left his academic career in 1980.

There is a non-confirmed version indicating that Inodoro lives in Argentina, in the country (the *pampa*), working with wild horses.

20 years later, a group of Inodoro's followers started the construction of C5, a real version of Inodoro's dream.

The name *C5* comes from the Spanish *CCinco* that means *The InCo C Compiler* ( *InCo* is a trademark of the Instituto de Computación of Montevideo, Uruguay).

Today C5 is a well-known programming language by the undergraduate students at InCo and the examples of Inodoro's dream are now real programs in C5:

```
/* The list of  integer example */

DT_typedef struct IntL{
              int element;
              struct IntL * {0} next;
              }   *MyType;
main(){
    DPT dp;
    MyType  obj;
    int i;
    dp= DT_pair(MyType, obj);
    C5_scanf(dp);
    for(i=C5_lenSearch(dp,"element")-1;i>=0;i--)
        C5_printf(C5_idxSearch(i,dp,"element"));
    }

/* The array of string example */
DT_typedef char * element;
DT_typedef element  MyType[4];
main(){
    DPT dp;
    MyType  obj;
    int i;
    dp= DT_pair(MyType, obj);
    C5_scanf(dp);
    for(i=C5_lenSearch(dp,"element")-1;i>=0;i--)
        C5_printf(C5_idxSearch(i,dp,"element"));
    }
```

In this paper, we present the version 0.98 of the C5 compiler including the functions `C5_printf`, `C5_scanf`, `C5_lenSearch` and `C5_idxSearch`.

## 1.2 Dynamics in C

Polymorphic functions are a well known tool for developing generic programs.

For example, the function *pop* of the Stack ADT

$$pop: \ \forall\, T.\ Stack\ of\ T \ \rightarrow \ Stack\ of\ T$$

has a single algorithm that will perform the same task for any stack regardless of the type of its elements. In this case, we say that the *pop* algorithm is similar for different instantiations of $T$.

A more complex and powerful way to express generic programs are the functions with dependent type arguments (i.e., the type of an argument may depend on the value of another) that perform different tasks depending on the argument type. These functions may inspect the type of the arguments at run time to select the specific task to be performed.

The C `printf` and `scanf` functions are two widely used examples of this kind of generic programs that are defined for a finite number of argument types. As we will see later, the type of these useful functions cannot be determined at compile time by a standard C compiler.

Even more powerful generic programs are achieved when we extend the finite number of argument types to the entire type system. This class of generic functions can perform different tasks depending on the argument type extending its expression power to include generic programs like parser generators (a top paradigm in generic programming).

C5 is a superset of the C programming language. The extensions introduced in C5 are the notion of Dependent Pair Type (DPT) and that of a Type Initialization Expression (TIE).

These extensions provide C5 with dynamic type inspection at run time and attribute type definition.

C5 is a minimal C extension that express a wide class of generic programs where the functions `C5_printf` and `C5_scanf` presented in this paper are representative examples.

## 1.3 The type of `printf`

The C creators [15] warn about the consequences of the absence of type checking in the `printf` arguments:

> " ... printf, the most common C function with a variable number of arguments, uses information from the first argument to determine how many other arguments are present and what their types are. It fails badly if the caller does not supply enough arguments or if the types are not what the first argument says."

Let us see through the simple example in Figure 1 how `printf` works. The first argument of `printf`, called the *format string*, determines the type of the other two: the expressions `4s` and `6.2f` indicate that the type of the second

```
main(){
      double n=42.56;
      char st[10]="coef";
      printf("%4s %6.2f",st,n);
      }
```

Figure 1: A simple C `printf` example.

argument is an array of characters while the third argument is a floating point notation number.

In the case of `printf` and `scanf`, the types declared in the format string are restricted to atomic, array of character and character pointer types. There is also some numeric information together with the type declaration (`4` and `6.2` in our example) that defines the printing format of the second and third arguments. These numeric expressions (attributes) will be called Type Initialization Expressions (TIEs) in C5.

A standard C compiler cannot type check statically the second and third arguments of the example presented in figure 1 because their types depend on the value of the first one (the format string).

In functions like `printf` and `scanf`, expressiveness is achieved at a high cost: type errors are not detected and, as a consequence unsafe code is produced.

However, some C compilers (e.g. the `-Wformat` option in `gcc` [10]) can check the consistence of the format string with the type of the arguments of `printf` and `scanf`. In this case, the format argument is a constant string (readable at compile time) and the C syntax is extended with the format string syntax.

This is not an acceptable solution of the problem because the syntax of the format string is specific for the functions `printf` and `scanf`.

A better solution can be found in Cyclone [21], a safe dialect of C. In this case, the type of the arguments of `printf` and `scanf` is a *tagged union* containing all of the possible types of arguments for `printf` or `scanf`. These tagged unions are constructed by the compiler (*automatic tag injection*) and the functions `printf` and `scanf` include the code to check at run time the type of the arguments against the format string.

Similar results can be obtained with other polymorphic disciplines in statically typed programming languages such as finite disjoint unions (e,g, Algol 68) or function overloading (e.g. C++).

This kind of solution of the `printf` typing problem has the following restrictions:

- The consistency of the format string and the type of the arguments is checked at run time and
- the set of possible types of the arguments of `printf` and `scanf` is finite and included in the declaration (program) of the functions.

However, the concept of *object with dynamic types* or *dynamics* for short, introduced by Cardelli [6] [1] provides an elegant and general solution for the `printf` typing problem.

A *dynamics* is a pair of an object and its type. Cardelli also proposed the introduction in a statically typed language of a new datatype (`Dynamic`) whose values are such pairs and language constructs for creating a dynamic pair (`dynamic`) and inspecting at run time its type tag (`typecase`).

Figure 2 shows a functional program using the `typecase` statement where `dv` is a variable of type `Dynamics` constructed with `dynamic`, *Nat* (natural numbers) and `X * Y` (the set of pairs of type `X` and `Y`) are types to be matched against the type tag of `dv`, `++` is a concatenation operator, and `fst` amd `snd` return the first and second member of a pair.

```
typetostring(dv:Dynamics): Dynamics -> String
    typecase dv of
        (v: Nat) " Nat "
        (v: X * Y) typetostring(dynamic fst(v):X)
                   ++ "  *  "
                   ++ typetostring(dynamic snd(v):Y)
        else "??"
    end
```

Figure 2: The statement `typecase`

Tagged unions or finite disjoint unions can be thought of as *finite versions* of `Dynamics`: they allow values of different types to be manipulated uniformly as elements of a tagged variant type, with the restriction that the set of variants must be fixed in advance.

C5 offers a way to embed *dynamics* within the C language that follows the concepts proposed by Cardelli.

The goal of the C5 language is to experiment with generic programs based on functions with dependent arguments under the following conditions:

- the type dependency of the arguments is checked at compile time and
- the functions accept (and are defined for) arguments of any type.

# 2 The C5 extensions

*Dynamics* has been implemented in C5 as an abstract data type called DPT (Dependent Pair Type). Instead of the statement `typecase` there are a set of functions that construct DPT pairs, inspect the type tag and read or assign values.

Since the use of DPTs is limited to a special class of generic functions, there is a C5 statement called `DT_typedef` to declare valid type definitions for the DPT library.

The major difference of the DPT library with Cardelli's *Dynamics* is concerned with the communication between the static and the dynamic universes:

- In the case of *dynamics*, there is a pair constructor (`dynamic`) for passing a static object to the dynamic universe. The inverse operation –the `typecase` statement– is a selector that retrieves the dynamic object to the static universe provided it matches with a given static type.

- In the case of the DPT library, the constructor `DT_pair` is the `dynamic` counterpart, but nothing equivalent to `typecase` can be found in C5. The only way to inspect a DPT object is by using a generic object selector (`C5_gos`) that encodes the static C selectors into the dynamic universe. In other words, it is easy to transfer a static object to the dynamic universe but the inverse is limited to atomic types. In compensation, it is possible to do some *object processing* within the dynamic universe.

This difference allows C5 to construct new dynamic objects at run-time without the *Dynamics* type checking requirements.

## 2.1 Dependent pairs in C5

For the sake of readability, we will simplify the C type system to `int`, `double`, `char` , `struct` , `union`, array, pointer, defined and function types.

The following is a brief introduction to the most important functions of the DPT library:

- `DPT DT_pair( C_Type t, t object)`
  The function returns a dependent pair where the type tag is the dynamic representation of the first argument `t` and the object member is a reference to the second argument `object`. The C5 compiler assures that DPTs are well formed by checking that the second argument is a variable whose type is the value of the first which is a `DT_typedef` type definition.

- `DPT C5_gos(DPT dp, int i)`
  The function is a universal selector for DPT pairs. If the type tag of `dp` is a struct or a union, then `C5_gos` yields a DPT pair with the type and value of the i*th* field. If `dp` is an array, then `C5_gos` returns a DPT pair with the type of the array elements and the i*th* element of the array. If `dp` is a pointer or `DT_typedef` DPT, then `C5_gos(dp,1)` yields a DPT pair constructed from the type of the referenced object and the object itself respectively. If `i` is out of range, an dynamic pair with error information is returned.

  `C5_gos` is not defined for atomic or function types.

- `DPT C5_fapply(DPT functionp, DPT_List args)`
  If `functionp` is a function pointer, `C5_fapply` type checks the function against the argument list contained in `args`.

If type checking is successful, `C5_fapply` applies the function of `functionp` to the n arguments of `args` and returns a DPT with the result value. Otherwise, the returned DPT includes error information.

- `int C5_gtype(DPT)`
  The DPT library is defined for the following type classes:

  1. `INT` - The `int` type in C.
  2. `CHAR` - The `char` type in C.
  3. `DOUBLE` - The `double` type in C.
  4. `STRUCT` - The set of C `struct` types.
  5. `UNION` - The set of C `union` types.
  6. `ARRAY` - The set of $Ctype[C\_expr]$ types.
  7. `POINTER` - The set of $* Ctype$ types.
  8. `TYPEDEF` - The set of `DT_typedef` type definitions.
  9. `FUNCTION` - The set of function types.

  Note that the C types `unsigned`, `short long`, `float`, `void`, and `enum` are not included.

  The function `C5_gtype` yields the type class of the dynamic pair argument.

- `int C5_isDUnion( DPT )`
  C Unions are not interesting for the DPT Library because there is no way to know the current field of an union at run time. For example, `C5_gos` cannot be defined for C unions.

  Instead of this kind of union , DPT functions recognize discriminated unions as a special case of the struct type.

  A struct with two fields where the first is an union and the second an integer is matched as a C5 Discriminated Union. In this case, the integer field is supposed to keep the information about the current field of the union.

  The function `C5_isDUnion` returns 1 when the type of a dynamic pair is a Discriminated Union. Otherwise returns 0.

- `int C5_isFunction(DPT)`
  In the C language, the declaration of function types cannot be directly expressed. Instead, we declare the type of function pointer.

  The function `C5_isFunction` returns 1 if the type of the argument is a function pointer. Otherwise returns 0.

- `int C5_gsize(DPT)`
  If the type tag of the argument is a struct, union or function the function returns respectively the field quantity, the size or the arguments number. If the tagged type is an atomic type `C5_size` returns 0, and in case of pointers or defined types the function returns 1.

- `char * C5_gname(DPT)`
  The function yields a string equal to the current type or field name of the type tag of the dynamic pair.

- `int C5_gpin(DPT)`
  The function returns the C5 pin number of the type member of the pair. Each C5 type has a unique pin number.

- `int C5_gint(DPT, int)`
  `double C5_gdouble(DPT, double)`
  `char C5_gchar(DPT, char)`
  `char *C5_gstr(DPT, char *)`
  These functions return the value of the pair if the type tag is respectively `int`, `double`, `char` and char pointer or array of char, In case of type mismatch the second argument is returned.

- `int C5_int_ass(DPT dp, int v)`
  `int C5_double_ass(DPT dp, double v)`
  `int C5_char_ass(DPT dp, char v)`
  `int C5_str_ass(DPT dp, char *v)`
  If the type tag of `dp` matches, these functions assign the value of the second argument to the second member of the first argument pair and the returned value is 1.

  In case of type mismatch no assigning is performed and the functions return 0.

The equivalence of the DPT library with *Dynamics* is showed in the following program which is a C5 version of the example presented in Figure 2:

```
void typetostring(DPT dv){
        switch(C5_gtype(dv)){
           case INT:    printf(" Int ");
                        break;"
           case STRUCT: if(C5_gsize(dv)==2){
                           typetostring(C5_gos(dv,1));
                           printf(" * ");
                           typetostring(C5_gos(dv,2));
                           }
                        else printf(" ?? ");
                        break;
           default:     printf(" ?? ");
           }
        }
```

We will use DPTs to express the C5 version of `printf` with the form:

$$void\ C5\_printf(DPT)$$

where the format string of the C `printf` function is expressed by the dynamic type of the pair argument. Notice that in this version the type dependency of the argument is checked at compile time while the possible types of the argument are not fixed.

The program below is a first C5 approach to the C `printf` example presented in figure 1:

```
DT_typedef char String[5];
DT_typedef float Fnr;
main(){
      String st="coef";
      Fnr n=42.56;
      c5_printf(DT_pair(String,st));
      c5_printf(DT_pair(Fnr,n));
      }
```

Note that the declared types `String` and `Fnr` are the arguments of the function `DT_pair`.

This is not a complete version of `printf` because the numeric information of the format argument is absent.

## 2.2 DPT list and atomic DPT constructors.

The DPT library includes an ADT of DPT list and a set of atomic DPT constructors to simplify the use of DPTs:

- `DPT_list dpnil()`
  The null list constructor.
- `DPT_list dpcons(DPT, DPT_list)`
  The inductive list constructor.
- `int dpempty(DPT_list)`
  Returns 1 if the argument is a null list.
- `DPT dphd(DPT_list)`
  It returns the head of the list. If the argument is the null list, the function returns the null DPT.
- `DPT_list dptl(DPT_list)`
  It returns the tail of the list. If the list is empty returns the null list.
- `int dplen(DPT_list)`
  It returns the length of the list.
- `DPT_list dpappend(DPT ,DPT_list)`
  It appends the DPT argument to the end of the list.

- `DPT dp_In(int)`
  `DPT dp_Ch(char)`
  `DPT dp_Do(double)`
  `DPT dp_St(char * )`
  The functions construct dynamic pairs using predefined types and the value of the argument. For example, `dpIn(124)` is equivalent to the following C5 code:

```
DT_typedef int IntType;
...
IntType vn=124;
DT_pair(IntType,vn);
```

The next example presents a DPT list constructed with elements of different types:

```
dpcons(DT_pair(DPT,dp_Ch('A')), dpcons( dp_Do(0.57),
        dpcons( dp_St("Hello"), dpnil())));
```

Note that the first element of the list is a dynamic containing a dynamic. `DPT` and `DPTi_list` are predefined types of the DPT library.

## 2.3 The Type Initialization Expression (TIE)

A TIE is a DPT list attached to a C5 type.

The syntax of a TIE is a comma-separated sequence of DPTs enclosed by brackets.

Constant expressions of atomic types do not need DPT constructors in a TIE declaration.

For example, the TIE { `'A', 0.57, "Hello"` } is correct and is translated by the C5 compiler to

```
dpcons(dp_Ch('A'),dpcons(dp_Do(0.57),dpcons(dp_St("Hello"),dpnil())));
```

This TIE declaration is equivalent to the TIE { `dp_Ch('A'), dp_Do(0.57), dp_St("Hello")` }.

There is a simple syntactical rule for inserting TIEs into a type declaration:
*a TIE is placed on the right of the related type.*

The next example shows two type definitions with TIEs:

```
DT_typedef int{1} Numbers[10]{2} [20]{3};
DT_typedef struct{
                  Numbers{4} nrs;
                  char{5} *{6} String_ptr;
                  }{7} Rcrd;
```

In the first type definition, the TIE {1} is attached to an `int` type and the TIEs {2} and {3} are attached to a double array. In the second definition, the TIEs

{4}, {5}, {6} and {7} are attached to the types `Numbers`, `char`, pointer of `char` and `struct` respectively.

TIEs can be inspected at run time using the following functions of the DPT library:

- `DPT_list C5_gtie(DTP)`
  It returns the DPT list in order to be directly manipulated. If the dynamic pair has no TIE, the null DPT list is returned.

- `int C5_gTIE_length(DPT)`
  the function returns the size of the TIE of the type tag of the dependent pair argument. If the TIE does not exist, the function returns 0.

- `int C5_gTIE_type(DPT, int idx)`
  the function applies `C5_gtype` to the TIE element indexed by `idx`. If the TIE does not exist, the function returns 0.

- `int C5_gTIE_int(DPT, int, int)`
  `double C5_gTIE_double(DPT, int, double)`
  `char C5_gTIE_char(DPT, int, char)`
  The functions yield the value of the TIE element indexed by the second argument. If the TIE element to be read does not exist, the function returns the third argument. In case of type mismatch a warning message is printed.

- `int C5_TIE_ass(DPT dp, int, DPT tieval)`
  The function assigns the value of `tieval` to the TIE of `dp` indexed by the second argument. If the assignment is successful the returned value is 1. If the TIE does not exist or the index is out of range or in case of type mismatch a warning message is printed.

After the introduction of TIEs, the C `printf` example presented in figure 1 can be completely expressed in C5 as follows:

```
DT_typedef char String[5] {4};
DT_typedef float {6,2} Fnr;
main(){
      String st="coef";
      Fnr n=42.56;
      c5_printf(DT_pair(String,st));
      c5_printf(DT_pair(Fnr,n));
      }
```

The TIEs {4} and {6,2} are respectively attached to the array and `float` types. Notice that TIE declarations are optional: in this program the `char` type of the first type definition has no TIE.

## 2.4 C5 Type equality.

In most cases, C5 functions (e.g. `C5_fapply`) require structural type equality.

In the next type definitions, the types `T1` and `T2` are not equally defined.

```
DT_typedef struct AT{
              int nr1, nr2;
              struct{ char *String; struct AT *link; } ST;
              } * T1;

DT_typedef int Integer;
DT_typedef struct BT {
              Integer cod;
              int age;
              struct{ char * name; struct BT * next; } ns;
              } * T2;
```

However, they are structurally equal ( `struct{int,int, struct{char ptr, rec ptr}}ptr`). This is the kind of equality used by C5 functions.

The structural equality of C5 types is checked by the `C5_type_seq` function. It returns 1 if the types of two dynamic pairs are at least structurally equal: Otherwise, the function returns 0.

```
int C5_type_seq(DPT d1, DPT d2){
      if(C5_gpin(d1)==C5_gpin(d2)) return(1);/* identical types */
/*  skip typedefs */
if(C5_gtype(d1)==VTYPEDEF)
            return(C5_type_seq(C5_gos(t1,1),t2));
if(C5_gtype(d2)==VTYPEDEF)
            return(C5_type_seq(t1,C5_gos(t2,1)));
      if(C5_gtype(d1)==C5_gtype(d2))a  /* same type class */
          switch(C5_gtype(d1)){
          case INT: case CHAR: case DOUBLE: return(1);
          case STRUCT:  case UNION:
               if(C5_gsize(d1)==C5_gsize(d2)){
                     int i;
                     for(i=1;i<=C5_gsize(d1);i++)
                           if(C5_type_seq(C5_gos(t1,i),
                                 C5_gos(t2,i))==0) return(0);
                      return(1);
                      }
               return(0);
          case VARRAY:
               if(C5_gsize(d1)==C5_gsize(d2))
                      return(C5_type_seq(C5_gos(t1,0),
                            C5_gos(t2,0)));
               return(0);
          case POINTER:
               if(C5_rec_ptr(t1) && C5_rec_ptr(t2)){
                     int p1=
```

```
                        is_ptr_checked(C5_gpin(t1),ptr_table);
                    int p2=
                        is_ptr_checked(C5_gpin(t2),ptr_table);
                    if(p1==NChk || p2==NChk) /* not checked */
                          return(C5_type_seq(V_gfield(t1,1),
                                V_gfield(t2,1)));
                    if(p1==Chkd && p2==Chkd) /*both checked*/
                          return(1);
                    }
else if(!C5_rec_ptr(t1) && !C5_rec_ptr(t2))
                    return(C5_type_seq(C5_gos(t1,1),
                          C5_gos(t2,1)));
               return(0);
           default:return(0);
           }
       return(0);
       }
```

Note that in the case of pointers, the function avoids infinite loops by using a table to assure that recursive pointers are checked once.

# 3 A generic version of `printf`

Since `C5_printf` accepts type expressions (DPTs) as arguments, it is straightforward to extend the restricted argument types of C `printf` (strings and atomic types) to the entire C type system.

For example, the type definition with TIEs presented in Figure 3 is an acceptable argument for the `C5_printf` function.

```
DT_typedef struct{
                  char ref[12];
                  double {2,3} *coef;
                  struct{
                        char name[40];
                        int {5} box_nrs[3];
                        } client;
                  } Client_Record;
```

Figure 3: A type definition with TIEs.

The next program shows a simplified and verbose version of the `C5_printf` function defined for the `int`, `double`, `char`, `struct`, `DT_typedef`, pointer and array types.

```
void C5_printf(DPT dp){
    int i;
    char format[100];
```

```
switch(C5_gtype(dp)){
    case INT:
        sprintf(format,"%%%dd",C5_gTIE_int(dp,0,6));
        printf(format,C5_gint(dp,0)); break;
    case DOUBLE:
        sprintf(format,"%%%d.%df",C5_gTIE_int(dp,0,6),
                                  C5_gTIE_int(dp,1,6));
        printf(format,C5_gdouble(dp,0.0)); break;
    case CHAR: printf("%c",C5_gchar(dp,'!')); break;
    case STRUCT:
        printf("\n struct %s={ ",C5_gname(dp));
            for(i=1;i<=C5_gsize(dp);i++){
                printf(" ");
                C5_printf(C5_gos(dp,i));
                }
        printf("}\n"); break;
    case ARRAY:
        printf("\n array %s=[ ",C5_gname(dp));
            for(i=0;i<C5_gsize(dp);i++){
                if(C5_gtype(C5_gos(dp,i,ErrorDp))==CHAR)
                    if(C5_gchar(C5_gos(dp,i),'!')=='\0')
                        break;
                 else if(i>0) printf(" ,");
                 C5_printf(C5_gos(dp,i));
                 }
            printf(" ]\n"); break;
    case POINTER: case TYPEDEF:
        C5_printf(C5_gos(dp,1)); break;
    }
}
```

The following `C5_printf` example prints an object of the type `Client_Record` presented in Figure 3:

```
main(){
      Client_Record cr;
      double r=2.8672;
      strcpy(cr.ref,"0037731443");
      cr.coef=&r;
      cr.client.box_nrs[0]= 1204;
      cr.client.box_nrs[1]= 82761;
      cr.client.box_nrs[2]= 464;
      strcpy(cr.client.name,"Carlos Gardel");
      C5_printf(DT_pair(Client_Record,cr));
      }
```

with the following result:

```
struct Client_Record={
array ref=[ 0037731443 ]
```

```
2.867
struct client={
array name=[ Carlos Gardel ]
array box_nrs=[  1204 ,82761 ,  464 ]
}
}
```

There is also a C5 version of `fprintf`

$$int\ C5\_fprintf(FILE\ *\ ,\ DPT)$$

# 4 A generic version of `scanf`

The `scanf` function of the C language scans input according to the format string argument which specifies the type and conversion rules of the other arguments. The types specified in the format argument are restricted to (references to) atomic and string types. The results from these conversions are stored in the arguments of the function.

As we did with `printf`, we introduce a generic version of `scanf` in C5:

$$DPT\ C5\_scanf(DPT)$$

where the format string of the C `scanf` function is expressed by the dynamic type of the DPT argument.

`C5_scanf` interprets the dynamic type of the argument as the grammar for parsing the input and, if the parsing is successful, the object member of the argument pair is constructed accordingly to the input. If the input cannot be parsed, `C5_scanf` returns a dependent pair with information about the error.

The resulting program includes a parser generator that can be compared with Yacc [14] and a scanner like Lex [17].

There is also a counterpart of C `fscanf` in C5:

$$DPT\ C5\_fscanf(FILE\ *\ ,\ DPT)$$

We introduce the `C5_scanf` function by first explaining the *lexical* meaning of the C types that belong to the lexical analyzer and then the *grammatical* meaning of the types related to the syntax analyzer.

## 4.1 The lexical analyzer

Atomic and string types are the *lexical* or *token* elements of `C5_scanf`. The actual version of `C5_scanf` accepts the following *lexical* types: `int`, `double`, `char`, character pointer and array of characters.

These types are interpreted in `C5_scanf` as follows:

- `int` is interpreted as the regular expression (RE) `[0-9]+`. If the type is attached with { `Signed`} then the RE is `[+-]?[0-9]+`.

- `double` is interpreted as the RE `[0-9]+.[0-9]+`. If the type is attached with { `Signed`} then the RE is `[+-]?[0-9]+.[0-9]+`.
- `char` {`ch`} will match a character equal to `ch`.
- `char A[N]` {`Word`} will match a string equal to `Word` if its length is less than `N` and starts with a letter or punctuation char followed of printable (excluded space) chars. An error is reported if no TIE is declared.
- `char *`{`RE`} will match the input according with the regular expression `RE`. If the TIE is absent the default RE is `[A-Za-z][A-Za-z0-9_]*`.

`C5_scanf` uses *token* type declarations to construct a regular expression table (in the Lex style) with the following order:

1. arrays of chars
2. characters
3. character pointers, `double` and `int` numbers.

There are also special functions to extend the table with comments and spacing characters. The default table has no comments and the spacing characters are $'\backslash r', '\backslash t', '\ '$ and $'\backslash n'$.

In case of ambiguous specifications, `C5_scanf` chooses the longest match. If there are more than one RE matching the same number of characters, the RE found first in the table is selected.

The example below shows how a string can be scanned according to the RE `[AB]+`:

```
DT_typedef char * {"[AB]+"} AB;
main(){
      AB ab;
      addComment("/*","*/");
      C5_printf(C5_scanf(DT_pair(AB,ab)));
      }
```

The function `addComment` enables comments with the declared start and ending strings. The program accepts the following input

```
AABBBAAAA  /* A C5_scanf example */
```

and the output will be

```
"AABBBAAAA"
```

The next input string

```
AA12xy   /* this string is not acceptable by the scanner */
```

cannot be parsed and therefore the output is an error message:

```
struct ErrorMessage={ "Syntax error"
 struct near_at_line={ "AA"      1 }
 }
```

## 4.2 The syntax analyzer

The types with a *syntactic* meaning in `C5_scanf` are: structures, arrays (array of char is excluded), type definitions , discriminated unions, pointers (char pointer is excluded) and recursive declarations.

### 4.2.1 Structures and arrays

A `struct` or an array type is a sequence of syntactic or lexical types. The set of strings accepted by this grammar (type) is the cartesian product

$$< S_0,\ S_1,\ \ldots\ , S_n >$$

where $S_0, S_1, ..., S_n$ are the sets of strings of the fields or elements of a given structure or array respectively.

### 4.2.2 Pointers and definition types

The set of strings accepted by pointer and definition types are the same than the referenced and the defined type respectively.

The next program shows a type (grammar) that includes the structured, pointer and defined types:

```
DT_typedef double Real;
DT_typedef struct{ int n; Real r; } *IntReal[2];
main(){
      IntReal ir;
      C5_printf(C5_scanf(DT_pair(IntReal,ir)));
      }
```

For example, the string `"123 0.432 21 0.55"` is an acceptable input for this program.

### 4.2.3 Discriminated unions

C unions cannot be used to express alternative grammars because they are not discriminated, that is, the compiler does not know which field of the union is currently stored.

By convention, we will represent alternative grammars in `C5_scanf` by the following type:

$$\begin{array}{l} DT\_typedef\ struct\{ \\ \qquad union\{\ d_0, .., d_i, .., d_n\ \}\ < id >; \\ \qquad int\ < id >; \\ \qquad \}\ < id >; \end{array}$$

where $d_0, .., d_i, .., d_n$ are the fields of the union and the integer field is called the *union discriminator* and is supposed to keep the information about the current field of the union. Thus, the discriminator field has no grammatical meaning.

The discriminated union type represents in `C5_scanf` the union of the sets of strings accepted by the fields (grammars) $d_0, .., d_i, .., d_n$.

### 4.2.4 The empty rule

The concept of empty rule is implemented in two ways:

1. **explicit way**

   The empty rule is a special field in discriminated unions. the *empty* fiels is a nullable *token* called `emptyProd` which is defined as follows:

   ```
   DT_typedef char {'\0'} emptyProd;
   ```

2. **implicit way**

   The empty rule is an alternative in fields of recursive pointer including the TIE `{0}`:

   ```
   struct NODE * {0} next;
   ```

   If the empty rule is matched, the pointer is assigned with the standard C `NULL` value.

This implementation is based on the proposal of Aycock and Horspool [4].

### 4.2.5 Recursive declarations

Recursive type declarations of discriminated unions allow us to express unbounded sets of strings.

For example, the program below accepts sequences of numbers and the constructed object will be a linked list of integers:

```
DT_typedef struct IntL{
              union{
                   int n;
                   struct{ struct IntL *next; int n; } RecProd;
                   } UU;
              int discriminator;
              } * Int_List;
main(){
      Int_List il;
      C5_printf(C5_scanf(DT_pair(Int_List,il)));
      }
```

Another version of this program can be done with the standard C declarations of linked lists:

```
DT_typedef struct IntL{
              int n;
              struct{ struct IntL * {0} next; int n; } RecProd;
              } * Int_List;
main(){
      Int_List il;
      C5_printf(C5_scanf(DT_pair(Int_List,il)));
      }
```

Note the TIE {0} in the recursive pointer. In this case `C5_scanf` uses the hidden discriminated union of the C `NULL` value for null pointers.

## 4.3 BNF notation

In most parser generators, grammars are expressed in BNF (Backus-Naur notation) or EBNF (Extended BNF).

The following example is a BNF grammar in Yacc syntax:

```
exp      :  NUMBER
         |  exp '+' exp
         ;
```

where `exp` is a nonterminal symbol and `NUMBER` and `'+'` are terminals (tokens). In `C5_scanf`, this BNF grammar can be expressed by the next type declaration:

```
DT_typedef struct EXP{
             union{
               int number;
               struct{
                 struct EXP *e1; char{'+'} pl; struct EXP *e2;
                 } RecP;
               } UU;
             int discriminator;
             } *exp;
```

## 4.4 The parsing algorithm

The algorithm of the `C5_scanf` parser generator is an implementation of the Earley algorithm [9] with a lookahead of $k = 1$. This algorithm is a chart-based top-down parser that accepts the complete set of context free grammar (CFG) and avoids the left-recursion problem.

The algorithm runs in $O(n^3)$ time order where $n$ is the number of symbols to be parsed.

The algorithm has been modified to construct an object of the type that represents the grammar. This is done by programming the recognizer so that it builds an object during the recognition process.

`C5_scanf` will produce parsers even in the presence of conflicts. There are some disambiguating rules in the Yacc style. For instance, the `if-else` and the arithmetic expression conflicts are solved in `C5_scanf`.

### 4.4.1 The if-else conflict

The program below is an example of the `if-else` conflict in `C5_scanf`:

```
DT_typedef  char Else[5] {'e','l','s','e'};
DT_typedef  char If[3]   {'i','f'};
DT_typedef struct IFE{
              union{
                char {'e'} exp;
                struct{ If i; struct IFE *e; } If_stmt;
                struct{ If i; struct IFE *e1;
                        Else s; struct IFE *e2;} If_Else_stmt;
                } UU;
              int  discriminator;
              } * Stat;
main(){
      Stat il;
      C5_printf(C5_scanf(DT_pair(Stat,il)));
      }
```

The input `if if e else e` produces two possible outputs for the same input `if (if e else e)` and `if (if e) else e`.

The ambiguity is detected by `C5_scanf` returning a diagnostic message:

```
C5_scanf: Disc. union "Stat" ambiguous in
   field 3 "If_Else_stmt" and
   field 2 "If_stmt".
   Suggestion: attach an int TIE to the "Stat" discriminator
   specifying the preferred alternative ({3} or {2}).
```

If we attach the TIE {2} to the discriminator field of `Stat` then the ambiguity is solved and the output will be

```
struct If_stmt={
 array If=[ if ]
 d_union Stat={
  struct If_Else_stmt={
   array If=[ if ]
   d_union Stat={ e}
   array Else=[ else ]
   d_union Stat={ e}
   }
  }
 }
```

### 4.4.2 Arithmetic expressions

The next token declaration in Yacc:

```
%left  '+'  '-'
%left  '*'  '/'
```

describes the precedence and associativity of the four arithmetic operators. The four tokens are left associative, and plus and minus have lower precedence than star and slash.

The next type declaration is the `C5_scanf` version of the above Yacc token declaration:

```
DT_typedef char {'+'} PLUS;
DT_typedef char {'-'} MINUS;
DT_typedef char {'*'} TIMES;
DT_typedef char {'/'} DIV;

DT_typedef PLUS   {LeftAss, 1} Plus;
DT_typedef MINUS  {LeftAss, 1} Minus;
DT_typedef DIV    {LeftAss, 2} Div;
DT_typedef TIMES  {LeftAss, 2} Times;
```

These disambiguating rules are declared in TIEs attached to type definitions related to token (or lexical) types. The first and second members of the TIE are respectively the associative and precedence rules.

## 4.5 Semantic Actions

Semantic actions in the Yacc style are simulated using the function `C5_compil`.

The nexr program shows the use of an action TIE in a simple grammar:

```
DT_typedef struct{
                  char {'<'} l;
                  char *id;
                  char {'>'} r;
                  } {"id"} IdExp[2] {1};
main(){
      IdExp ie;
      C5_printf(C5_compil(C5_scanf(DT_pair(IdExp,ie))));
      }
```

The TIE {`1`} selects the second element of the array and { `"id"` } selects the second field of the structure.

For example, this program accepts the string `" < one > < two > "` and the output is `"two"`.

## Programming with `C5_scanf`.

The following C5 programs are three motivating examples that illustrate the use of the `C5_scanf` function.

### Matrix

The example below prints an element of a $2 \times 3$ matrix constructed by `C5_scanf`:

```
DT_typedef int Matrix[2][3];
main(){
      Matrix mtx;
      if(C5_scanfError(C5_scanf(DT_pair(Matrix, mtx)))
            printf("Cannot read the matrix.\n");
      else printf("mtx[1][2]=%d\n",mtx[1][2]);
      }
```

Notice the way the variable `mtx` is used to communicate the dynamic and the static universes. This is an useful programming methodology in C5: the user constructs an object in the dynamic universe which is *processed* in the static universe.

### A desk calculator

The next program shows a desk calculator that includes associative and precedence rules to avoid ambiguous grammars:

```
     /*  Tokens  */
DT_typedef char {'+'} PLUS;
DT_typedef char {'-'} MINUS;
DT_typedef char {'*'} TIMES;
DT_typedef char {'/'} DIV;

   /*  C5 functions   */
DT_typedef int Number;
Number c5_Add( Number a, Number b){ return(a+b); }
Number c5_Mul( Number a, Number b){ return(a*b); }
Number c5_Dvd( Number a, Number b){ return(a/b); }
Number c5_Sub( Number a, Number b){ return(a-b); }
Number c5_Umi( Number a ){ return(-a); }

 /* The grammar and semantic actions  */
#define Ae_ struct Aexp *
DT_typedef struct Aexp{
            union{
              Number number;
              struct{ char {'('} lp; Ae_ e; char {')'} rp;}
                            {"e"}                        parProd;
              struct{ Ae_ e1; PLUS  {LeftAss, 2} add; Ae_ r1;}
                            {dp_Fn(c5_Add),"e1","r1"}   addProd;
              struct{ Ae_ e2; TIMES {LeftAss, 3} times; Ae_ r2;}
                            {dp_Fn(c5_Mul),"e2","r2"}   mulProd;
              struct{ Ae_ e3; MINUS {LeftAss, 2} minus; Ae_ r3;}
                            {dp_Fn(c5_Sub),"e3","r3"}   subProd;
              struct{ Ae_ e4; DIV {LeftAss, 3} div; Ae_ r4;}
```

```
                        {dp_Fn(c5_Dvd),"e4","r4"}   dvdProd;
            struct{ MINUS {LeftAss, 4}  um; Ae_ e; }
                        {dp_Fn(c5_Umi),"e"}      uminusProd;
            } uu;
            int disc;
          } *AritihmeticExp;
main(){
        AritihmeticExp aexp;
        addComment("||","\n");
        C5_printf(C5_compil(C5_scanf(DT_pair(AritihmeticExp,aexp))));
        }
```

For example, this calculator accepts the input `10 + 2 * 4 / - 2 - 2` and produces the output `4`.

### XML checker.

The example below shows a partial and simplified version of a well-formed XML document checker.

```
DT_typedef char *{"[^<&>]+"} charD;
DT_typedef struct{ char {'<'} l; char *id; char {'>'} r; } {"id"} STag;
DT_typedef struct{ char l[3] {"</"}; char *id; char {'>'} r;} {"id"} ETag;
DT_typedef struct{ char {'<'} l; char *id; char r[3]{"/>"};} EmptyElemTag;

DT_typedef struct{
                union{ charD chd; char * id; } UU;
                int discriminator;
                } CharData;

DT_typedef  struct CharDL{
                union{
                        emptyProd nil;
                        struct{ struct CharDL *c; CharData cd;} CDls;
                        } DU;
                int discriminator;
                } *CharDataList;

DT_typedef  struct{
                CharDataList cdl;
                struct XML_EL_LS *els;
                } {"els"}  XMLcontent;

int c5_cmpstr( char * s1, char * s2, int rec){
        if(strcmp(s1,s2))
                printf("Error: incorrect nested tags %s %s.\n",s1,s2);
        return(rec);
        }

DT_typedef  struct XML_EL{
```

```
                union{
                        EmptyElemTag  eet;
                        struct{ STag start; XMLcontent c; ETag end; }
                                {dp_Fn(c5_cmpstr),"start","end","c"} elem;
                        } DU;
                int discriminator;
                } *XMLelement;

DT_typedef  struct XML_EL_LS{
                union{
                        emptyProd {0} nil;
                        struct{ struct XML_EL_LS *next; struct XML_EL *el;}
                                        {"next","el"}  els;
                        } DU;
                int discriminator;
                } *XMLelementList;
main(){
        XMLelement xmldoc;
        C5_compil(C5_scanf(DT_pair(XMLelement,xmldoc)));
        }
```

This program accepts the following XML document

```
<message>
     <to>juanma@adinet.com</to>
     <from>marcos@adinet.com</from>
     <subject>XML test </subject>
     <text>
        --Can you check this with C5_scanf? ...
     </text>
 </message>
```

However, it rejects this input text with incorrect nested tags:

```
<message>
    <subject>  XML test of nested tags.  </message>
</subject>
```

Notice that in the case of a successful check, the variable `xmldoc` contains a structured XML document that can easily be inspected or processed.

# 5 Searching values in a DPT pair.

The DPT library includes functions to search values by their type identifiers:

- `int C5_lenSearch( DPT , char * ident )`
  The function returns the number of `ident` occurrences in the first argument.

- `DPT c5_idxSearch( int idx, DPT , char * ident )`
  The function returns the dynamics of the $idx^{th}$ occurrence of `ident` in the first argument. 0 is the first occurrence of `ident`. A null dynamics is returned if `idx` is not valid.

The functions follow the next searching rules:

1. the functions check the name of the current dynamic type. In case of non-atomic types, the functions may inspect the type expression.
2. `int`, `double`, `char`, char pointer, array of char and functions are atomic objects.
3. array elements are recursively inspected from 0 to the upper bound.
4. structure fields are recursively inspected from the first to the last.
5. C5 discriminated unions are inspected and evaluated according to the discriminator value.
6. Type definitions and non-null pointers are evaluated to the defined and referenced object respectively.

## 5.1 Examples

### 5.1.1 Type name count.

The next program prints `3` for the input `w1 w2` :

```
DT_typedef char * Words[2];
main(){
  Words ws;
  printf(" len=%d\n",C5_lenSearch(
      C5_scanf(DT_pair(Words,ws)),"Words"));
  }
```

As this example shows, type names may be shared by different types.

`C5_lenSearch` founds first the array `Words` and then two strings that are also identified with the type name `Words` are founded when the function inspects the array elements.

### 5.1.2 Reverse order.

The next program reads words from the input and prints them in reverse order:

```
DT_typedef char * {"[^ ]+"} element;
DT_typedef struct IntL{
              element el;
              struct IntL * {0} next;
              }   *MyType;
```

```
main(){
      DPT dp;
      MyType  obj;
      int i;
      dp= DT_pair(MyType, obj);
      C5_scanf(dp);
      for(i=C5_lenSearch(dp,"element")-1;i>=0;i--)
              C5_fprintf(stdout,C5_idxSearch(i,dp,"element"));
      }
```

The input `1 two --- |||| five5 6` produces the output `6 five5 |||| --- two 1`.

We can replace the linked list declaration by a matrix and the program still works:

```
DT_typedef char * {"[^ ]+"} element;
DT_typedef   element MyType[2][3];
main(){
      DPT dp;
      MyType  obj;
      int i;
      dp= DT_pair(MyType, obj);
      C5_scanf(dp);
      for(i=C5_lenSearch(dp,"element")-1;i>=0;i--)
              C5_fprintf(stdout,C5_idxSearch(i,dp,"element"));
      }
```

Likewise the previous example the input `1 two --- |||| five5 6` produces the output `6 five5 |||| --- two 1`.

# 6 About the C5 compiler (version 0.98, September 2006).

The C5 compiler has been developed in 1999 at the *Instituto de Computación (InCo)* in Montevideo, Uruguay. The compiler translates C5 programs into C code.

The C5 parser is a extended C parser with few grammatical modifications. The compiler consists on about 5500 lines where 900 of them are the actual type checker. The compiler parses C5, does type checking of DPT constructors and translates the generated syntax tree to C code.

In case of a successful compilation, C5 produces three C files:

1. `C5_defs.h`
   Type definitions and `extern` declarations.

2. `C5_out.c`
   It includes the functions `C5_gos` and `C5_fapply`, and the type database required by the DPT library.

3. `C5_prog.c`
   This file has the translated C5 source code.

The C5 type checker is simple: for every `DT_pair` invocation C5 checks statically if the first argument is a `DT_typedef` type definition and if the second is a variable of the same type than the value of the first argument.

The current implementation of C5 (Version 0.98, September 2006) with a sample of C5 programs can be found on the Web at

`http://www.fing.edu.uy/~jcabezas/c5`

`C5_scanf` has been implemented in C5 and consists on about 2300 lines where 700 of them belong to the lexical analyzer.

# 7 Related work

The statically typed programming languages Amber [6] and Modula-3 [7] include notions of a dynamic type and a typecase statement. This idea can also be found in functional programming [16] [20] [8] and in type-safe C dialects like Ccured [18] where *dynamics* are used for converting C in a type safe language.

Although C5 may assign accurate types to untyped C programs like `printf`, it is not a type-safe C dialect but rather a C-based framework for generic programming.

In our knowledge, C5 is the first C extension with dynamics developed for generic programming.

The extension of functional languages with dependent types is another interesting alternative for generic programming: Cayenne [2] –a Haskell-like [12] language with dependent types– is powerful enough to encode predicate logic at the type level and thus express generic functions like `printf` without restrictions.

In a close research line to dependent types, the Generic Programming community [5][11]. is developing another approach. PolyP [13] is an example of this work that achieves an expressive power similar to that of dependent types by parameterizing function definitions with respect to data type signatures.

## 7.1 `C5_scanf`

Most parsers in use today are based on efficient linear-time algorithms that accept a subset of CFGs (LL,LR or LALR) [14].

The primary objection to the Earley's algorithm is not functionality but with its run-time response.

Nevertheless, the practical use of Earley parsing has become an interesting alternative in the last years: Accent [19] is the first Earley parser generator along the lines of Yacc and DEEP [3] is an efficient directly-executable Earley parsing.

Finally, we did not found parser generators that accept grammars denoted with C types in the `C5_scanf` style.

# 8 Conclusions

The generic functions `C5_printf` and `C5_scanf` show that a static typed language extended with DPTs (dynamics) and TIEs can be powerful enough to express a wide class of generic functions in a straightforward, compact and safe way.

Even though the communication between static and dynamic types is also restricted to avoid typing conflicts, we have not detected practical limitations when implementing generic functions like `C5_scanf`.

The C5 version of Inodoro's dream examples shows clearly the abstraction level that this language can achieve.

The improvements of version 0.98 of the C5 compiler have transformed `C5_scanf`, `C5_compil`, `C5_lenSearch` and `C5_idxSearch` in a powerful parsing framework.

The most remarkable property of `C5_scanf` is the object construction. The user just need to define a type. The parsing result is an object of that type. Thus, the user may inspect or process the resulting object using the `C5_compil`,`C5_lenSearch` and `C5_idxSearch` functions.

The parsing process becomes transparent to the user in `C5_scanf`. This is a good example of generic programming in C5.

# References


[1] Martin Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic Typing in a Statically Typed Language. In *16th POPL*, pages 213–227, 1989.

[2] Lennart Augustsson. Cayenne - a Language with Dependent Types. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, pages 239–250, USA, 1998. ISBN 0-58113-024-4.

[3] "John Aycock and Nigel Horspool". "directly-executable Earley parsing". *Lecture Notes in Computer Science*, 2027:229+, 2001.

[4] John Aycock and R. Nigel Horspool. Practical Earley Parsing. *The Computer Journal*, 45(6):620–630, 2002.

[5] R. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic Programming - An Introduction -. In *Advanced Functional Programming, LNCS 1608*. Springer-Verlag, 1999.

[6] Luca Cardelli. "amber". In Guy Cousineau, Pierre-Louis Curien, and Bernard Robinet, editors, *"Combinators and functional programming languages : Thirteenth Spring School of the LITP, Val d'Ajol, France, May 6-10, 1985"*, volume 242. Springer-Verlag, 1986.

[7] "Luca Cardelli, James Donahue, and Lucille Glassman". Modula-3 report (revised). Technical report, DEC SRC-RR-52, 1989.

[8] "James Cheney and Ralf Hinze". A Lightweight Implementation of Generics and Dynamics;. In *Haskell Workshop 2002*, October 2002.

[9] Jay Earley. An Efficient Context-Free Parsing Algorithm. *Communications of the ACM*, 13(2):94–102, February 1970.

[10] GNU. *Using and Porting the GNU Compiler Collection (GCC).* Free Software Foundation, http://gcc.gnu.org/onlinedocs/gcc-2.95.3/gcc.html.

[11] R. Hinze. Polytypic Programming with Ease. In *4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99), Tsukuba, Japan.*, Lecture Notes in Computer Science Vol. 1722, pages 21–36. Springer-Verlag, 1999.

[12] Paul Hudak, Simon L. Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairnbairn, Joseph H. Fasel, Maria M. Guzman, Kevin Hammond, John Hughes, Thomas Johnsson, Richard B. Kieburtz, Rishiyur S. Nikhil, Will Partain, and John Peterson. Report on the Programming Language Haskell, A Non-strict, Purely Functional Language . *SIGPLAN Notices*, 27(5):R1–R164, 1992.

[13] P. Jansson and J. Jeuring. PolyP - A Polytypic Programming Language Extension. In *POPL 97: The 24th ACM SIGPLAN-SIGACT Symposium of Principles of Programming Languages* , pages 470–482. ACM Press, 1997.

[14] Steven C. Johnson. Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979.

[15] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*, page pg. 71. Prentice Hall, 1977. ISBN 0-13-1101-63-3.

[16] Xavier Leroy and Michel Mauny. Dynamics in ML. *Journal of Functional Programming*, 3(4):431–463, 1993.

[17] Michael E. Lesk and Eric Schmidt. "lex: A lexical analyzer generator". In *UNIX Programmer's Manual*, volume 2, pages 388–400. Holt, Rinehart, and Winston, New York, NY, USA, 1979. AT&T Bell Laboratories Technical Report in 1975.

[18] "George C. Necula, Scott McPeak, and Westley Weimer". "CCured: type-safe retrofitting of legacy code". In *Symposium on Principles of Programming Languages*, pages 128–139, 2002.

[19] "Friedrich Wilhelm Schrer". The ACCENT Compiler Compiler. Technical report, GMD Report 101, 2000.

[20] Mark Shields, Tim Sheard, and Simon Peyton Jones. Dynamic Typing as Staged Type Inference. In *POPL 98: The 25th ACM SIGPLAN-SIGACT Symposium of Principles of Programming Languages*, pages 289–302, Jan 1998.

[21] Jim Trevor, Greg Morriset, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe Dialect of C. In *The USENIX Annual Technical Conference* , Monterrey , CA, 2002.

.