THE C5 PROGRAMMING LANGUAGE
2007 Edition

Juan José Cabezas

Preface

This is the 2007 Edition of the C5 programming language manual.

C5 is a superset of the C programming language developed at the *Instituto de Computación* (InCo). 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 present edition of the C5 manual describes the version 0.98 of the C5 compiler and a set of generic libraries created by the following projects:

- Functions in C5.
- Parsing in C5.
- The OPM machine.
- Generic fonts in C5.
- Typed Windows.

Like the previous editions, the 2007 Edition is a recompilation of the technical reports of the C5 projects published by the *InCo-PEDECIBA*.

The main differences between the 2007 Edition and the previous edition are:

1. Rewriting of the introduction chapter including Inodoro's dream.
2. A new chapter joining the equality, selection and copy functions.
3. A new chapter about the function type in C5.

4. A new version of `C5_scanf`.

5. Several errors of the previous edition detected by the students were corrected.

This manual is mainly used by the teams of the C5 projects and the students of the courses *Introducción a la Programación para Diseño Gráfico* and *Diseño de Compiladores* of the study programs *Ingeniería en Computación* and *Maestría en Informática* of PEDECIBA.

The support and suggestions of many colleagues and students have added greatly to the developing of C5 and the pleasant writing of this manual. In particular: Gustavo Betarte, Hector Cancela, Zelmar Echegoyen, Alberto Pardo, Pablo Queirolo, Bengt Nordstriöm and Alfredo Viola.

Special thanks to the approximately 400 computer engineering students at InCo who tested the successive versions of the C5 compiler.

Montevideo, April 20th, 2007.

.

Contents

Chapter 1

Introduction

27 years ago, Inodoro Pereira [1] –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* at Montevideo when his famous dream came up.

1.1 Inodoro's dream

> " 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 *next;
                } *MyType;
  main(){
        MyType ils;
        scanf(" %MyType ", ils);
        printf(" %MyType ", ils);
        }
```

[1]Our fictitious teacher is inspired on the great comics created by the Argentinian Roberto Fontanarrosa.

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 –of course– that the input matches 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 *next;
                   } *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) main 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 C Compiler of InCo* (*InCo* is a trademark of the *Instituto de Computación* at Montevideo, Uruguay).

Today, C5 is used by the students at InCo and the old examples of Inodoro's dream are now real programs:

```
/* 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"));
    }
```

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).

The other 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 2007 edition of the C5 programming language manual presents the version 0.98 of the C5 compiler.

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 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.

1.2.1 The type of `printf`

The C creators [22] 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 1.1 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

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

Figure 1.1: A simple C `printf` example.

the second 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 1.1 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` [13]) 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 [32], 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.

1.2.2 Dynamic types

However, the concept of *object with dynamic types* or *dynamics* for short, introduced by Cardelli [7] [2] 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 1.2 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 1.2: 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.

1.3 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.

1.3.1 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 instance, `C5_gos` cannot be defined for C unions.

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

  A C5 Discriminated Union is defined as a struct with two fields where the first is an union and the second a integer. 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 1.2:

```
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 1.1:

```
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.

1.3.2 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.

1.3.3 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 1.1 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.

1.4 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 1.3 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 1.3: 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 1.3:

```
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

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

Chapter 2

Equality, selection and copy functions.

Equality, selection or copy are type dependent in C. This means that specific operators (or functions) must be defined for each type. For instance, the `*` and `.` operators are selectors for the pointer and `struct` types respectively.

In this chapter, we introduce the generic functions `C5_typeSeq`, `C5_seq`, `C5_lenSearch`, `C5_idxSearch`, `C5_copy` and `C5_newdp`. The functions are members of the DPT library.

2.1 Type equality in DPT pairs.

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

The notion of structural equality states that two objects are of the same type if the two objects are the same at a structural level regardless of what their type names are.

For example, 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_typeSeq` function:

```
int C5_typeSeq(DPT dp1, DPT dp2)
```

It returns 1 if the types of two dynamic pairs are at least structurally equal. Otherwise, the function returns 0.

The function follows the next rules:

1. a TYPEDEF DPT is evaluated to the *definiens* before equality checking.
2. INT DPTs are equal.
3. CHAR DPTs are equal.
4. DOUBLE DPTs are equal.
5. STRUCT or UNION DPTs are equal if they have the same field number and all the fields (with equal index) are equal.
6. ARRAY DPTs are equal if the arrays have the same size and the type of the elements of the arrays are equal.
7. POINTER DPTs are equal if the referenced objects have equal types. Note that in the case of pointers, the function `C5_typeSeq` avoids infinite loops by using a table to assure that recursive pointers are checked once.
8. FUNCTION DPTs are equal if they have the same argument number and the arguments and the result type are equal.

Appendix D presents a formal specification of the C5 structural type equality.

2.2 Object equality in C5

A standard C compiler have equality operators for constant expressions, variables of atomic types and pointers. In case of structured types like `struct` or arrays, the programmer must define an equality function for each defined type.

The function `C5_seq` is a generic function for object equality:

`int C5_seq(DPT dp1, DPT dp2)`

It returns 1 if the objects are equal, otherwise the function returns 0. The function follows the rules:

1. the types of the pairs are equal (see `C5_typeSeq`).
2. a TYPEDEF DPT is evaluated to the definiens before equality check.
3. INT DPTs are equal if their values are equal.
4. CHAR DPTs are equal if their values are equal.
5. DOUBLE DPTs are equal if their values are equal.
6. STRUCT DPTs are equal if the fields of the first pair and the respective fields (same index) of the second argument are equal.
7. ARRAY DPTs are equal if the array elements of the first argument and the respective array elements (same index) of the second argument are equal.
8. POINTER DPTs are equal if the referenced objects are equal.
9. C5 discriminated unions DPTs are equal if the discriminator values are equal and the active field of the first and second argument are equal.
10. FUNCTION and UNION DPTs are not defined for this kind of equality.

Appendix D presents a formal specification of the C5 object equality.

2.3 Searching objects 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. Note that 0 is the first occurrence of `ident`. A null dynamic pair 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 inspect the type expression.
2. `int`, `double`, `char`, char pointer, array of char and functions are atomic objects.
3. array elements are inspected from 0 to the upper bound.
4. structure are inspected from the first to the last field.
5. C5 discriminated unions are inspected and evaluated according to the discriminator value.
6. Type definitions and non-null pointers are evaluated to the definiens and referenced object respectively.

2.3.1 Examples

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"));
  }
```

This example shows that type names may be shared by different types.

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

Reverse order.

The next program reads words from the input using `C5_scanf` 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`.

2.4 The copy function in DPT pairs.

The generic function `C5_copy` copies the values of DPT pairs:

```
int C5_copy(DPT cpy,DPT src)
```

The function is defined according to the following rules:

1. the type of the pairs are equal (see `C5_typeSeq`).
2. a TYPEDEF DPT is evaluated to the definiens.
3. INT DPTs. It copies the value of the source DPT to the copy DPT.

4. CHAR DPTs. It copies the value of the source DPT to the copy DPT.
5. DOUBLE DPTs. It copies the value of the source DPT to the copy DPT.
6. STRUCT DPTs. The function copies the fields of the source DPT to the respective fields of the copy DPT.
7. ARRAY DPTs. It copies the array elements of the source DPT to the respective array elements of the copy DPT.
8. POINTER DPTs copies the reference value of the source DPT to the copy DPT. Note that no memory allocation is performed when copying pointers.
9. C5 disciminated unions DPTs copies the active field of the source DPT to the respective field of the copy DPT. In this case, the discriminator value is also copied to the copy DPT.
10. FUNCTION and UNION DPTs are not defined for this function.

The `C5_copy` function does not allocate memory. In the case of pointers it copies reference values and in the case of static types the function supposes that the object has been previously constructed.

The function `C5_newdp` is specially appropriated for creating new fresh copies from a existing DPT:

```
DPT C5_newdp( DPT dp)
```

This function returns a new DPT that is a copy of the argument. Note that the new DPT is not connected to a C variable. The only way to construct a new DPT connected to a C variable is by using the function `DT_pair`.

2.4.1 Example

The next example shows a C5 program using `C5_copy` and `C5_newdp`:

```
DT_typedef struct {
              char ch;
              int n;
              } Struct;
main(){
    Struct st;
    DPT dp1, dpcopy, newdp;
st.ch='A';
st.n=222;
dp1= DT_pair(Struct,st);
dpcopy= DT_pair(Struct,st);
printf("\n dp1="); C5_fprintf(stdout,dp1);
C5_copy(dpcopy, dp1);
newdp= C5_newdp(dp1);
st.ch='Z';
st.n=-1;
printf("\n dpcopy="); C5_fprintf(stdout, dpcopy);
printf("\n newdp="); C5_fprintf(stdout, newdp);
}
```

The struct variable `st` is assigned after the copy of `dp1`.

The output of the program

```
dp1=A    222
dpcopy=Z      -1
newdp=A    222
```

shows that the copy created with `C5_newdp` does not share the memory with `dp1`.

Chapter 3

Functions in C5.

Functions are not first class members in the C language. It is not possible to declare a variable of function type or assign a function to variables. Instead, the C language accepts function pointers and this is the way functions are handled as objects in a C program.

Function types are also declared through pointer type definitions.

The function type in C5 transforms C functions in real first class members of the language.

In this chapter we present the function type in C5, the constructor `dp_Fn`, the selector `C5_fapply` and the generic function `C5_compil`.

3.1 The function type

Figure 3.1 presents a C program including the definition and construction of a function variable.

The version 0.98 (September, 2006) of the C5 compiler includes function pointers definitions for DPT construction so that the C program presented in Figure 3.1 can be expressed in C5:

```
  DT_typedef int (* FunctionType)( int , char );

  int my_func(int n, char c){
      if(c=='0') return(0);  else return(n);
      }

  main(){
DPT fdp;
FunctionType mf;
        mf= & my_func;
  typedef int (* FunctionType)( int , char );

  int my_func(int n, char c){
        if(c=='0') return(0); else return(n);
        }

  main(){
        FunctionType mf;
mf= &my_func;
printf("%d", &mf(123, '5'));
        }
```

Figure 3.1: A function type in C.

```
fdp= DT_pair(FunctionType, mf);
C5_printf(C5_fapply(fdp,
              dpcons(dp_In(123),dpcons(dp_Ch('5'),
                     dpnil()))));
}
```

Note the use of the function `C5_fapply)`. This is the only way to use (inspect, select) a function DPT.

3.1.1 `C5_fapply`

The function `C5_fapply` performs functional application in C5:

$$C5\_fapply: \; DPT \;\times\; DPT\_List \;\rightarrow\; DPT\_List$$

If the first argument is a function pointer, `C5_fapply` type checks (see the function `C5_type_seq`) the function against the argument list contained in the second argument of `C5_fapply`.

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

3.1.2 `dp_Fn`

However, when the name of a function starts with `c5`, it is possible to construct function DPTs avoiding the `DT_typedef` declaration. In this case the

C5 constructor `dp_Fn` obtains the type of the function from its signature to construct a DPT.

The constructor `dp_Fn` allows a compact C5 version of the C program presented in Figure 3.1:

```
  int c5_my_func(int n, char c){
      if(c=='0') return(0);  else return(n);
      }

  main(){
C5_printf(C5_fapply(dp_Fn(c5_my_func),
dpcons(dp_In(123),dpcons(dp_Ch('5'),
                        dpnil())))));
}
```

3.1.3 C5_compil

The function `C5_compil` is a good example to show the use of `C5_fapply` in generic programs.

$$C5\_compil : \ DPT \ \rightarrow \ DPT$$

The function is a generic translation program. The translation rules required by `C5_compil` are provided by the TIEs of the argument of `C5_compil`. In other words, the object to be translated carries its own translation rules.

If the type of a dynamic pair is atomic (`int`, `char`,`double`, `char *` or array of char) or its type name starts with "Token", `C5_compil` returns its argument without evaluation. Otherwise, the dynamic pair is evaluated as follows:

1. If the type member of the pair has a TIE, then `C5_compil` evaluates the TIE as follows:
   (a) the elements of the TIE are evaluated (in sequence, from the first to the last) and `C5_compil` returns the result of the first one.
   (b) if the first element of the TIE is a function, the remaining elements are supposed to be the arguments of the function and the output of `C5_fapply` is returned.
   (c) if a member of a TIE attached to a `struct` is a string (`char *`) then `C5_compil` compares the string with the field names of the structure. If the string matches, a evaluation of the matched field is performed. Otherwise, the result pair is the original string.

   (d) if a member of a TIE attached to an array is a integer with a value within the bounds of the array, `C5_compil` evaluates the indexed element. Otherwise, the result pair is the original integer.

2. In case of a pair of `struct` or array type without TIE, the result of the evaluation is the pair itself.

3. In case of discriminated unions, pointers or definitions with no TIEs, the result of the evaluation is the respective evaluation of the valid field, the referenced value or the defined object.

4. pairs of function type are returned without changes.

The specification and a simplified version of the C5 source code of `C5_compil` can be found in Appendix F.

3.2 Examples

The examples presented below show the use of `C5_compil` and C5 functions.

3.2.1 Field selector

The following example shows how `C5_compil` is used to select a certain value of a data structure. The TIE `id` selects the second field of the structure and the TIE {`1`} selects the second element of the array.

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

The program returns `"two"` for the input `< one > < two >`.

3.2.2 The sum of a integer list.

The next example shows how `C5_compil` uses a c5 function:

```
int c5_add(int number, int recProd){ return(number + recProd);}

DT_typedef struct IntL{
              int number;
              struct{
                      union{
                              emptyProd {0} nil;
                              struct IntL *next;
                              } UU;
                      int discriminator;
                      } recProd;
              } {dp_Fn(c5_add),"number","recProd"} *Word_List;
main(){
      Int_List nrls;
      C5_printf(C5_compil(C5_scanf(DT_pair(Int_List,nrls))));
      }
```

Note that the functional dynamic pair is constructed with `dp_Fn` and the arguments of the function are the fields `number` and `recProd` of the structure `Int_List`.

The TIE {0} in `emptyProd {0} nil;` forces `C5_compil` to return 0 when detecting an empty list.

`C5_printf` is the C5 generic print function and `C5_scanf` is a scanner that interprets the dynamic type of the argument as the grammar for parsing the standard input and, if the parsing is successful, the object member of the argument pair is constructed according to the input.

`C5_scanf` constructs a list of integers, `C5_compil` computes the sum of the list which is printed by `C5_printf`. For example, the input `11 22 33` produces the output `66`.

3.2.3 Word count.

The following example changes the type of the linked list of the previous example to *word* and the first argument of `c5_add` to the constant 1. .

```
int c5_add(int one, int recProd){ return(one + recProd); }

DT_typedef struct WordL{
              char * word;
              struct{
                      union{
```

```
                                emptyProd {0} nil;
                                struct WordL *next;
                                } UU;
                        int discriminator;
                        } recProd;
                } {dp_Fn(c5_add), 1 ,"recProd"} *Word_List;
main(){
      Word_List wls;
      C5_printf(C5_compil(C5_scanf(DT_pair(Word_List,wls))));
      }
```

For example, if the input of this program is `one two three`, it returns 3.

3.2.4 The reverse function.

The next example applies the simple `c5_reverse` function to the input list to print it in reverse order.

```
char * c5_reverse(char *word, char *recProd){
       printf("%s ",word);
       return(recProd);
       }

 DT_typedef struct WordL{
                char *word;
                struct{
                        union{
                                emptyProd {""} nil;
                                struct WordL *next;
                                } UU;
                        int discriminator;
                        } recProd;
                } {dp_Fn(c5_reverse),"word","recProd"} *Word_List;
 main(){
       Word_List wls;
       C5_compil(C5_scanf(DT_pair(Word_List,wls)));
       }
```

For example, if the input of this program is `one two three`, it prints `three two one`.

Chapter 4

Parsing in C5

Generic functions in C5 are powerful enough to express a parser generator in the Yacc style [21].

Yacc and Lex are software tools used to generate a parser in C code that can be compiled together with other C programs.

C5 has the required expressiveness to include a parser generator in the DPT Library making the creation of a parser transparent for the C5 programmer.

In this chapter we introduce the parsing functions `C5_scanf` and `C5_fscanf`.

4.1 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 [21] and a scanner like Lex [24].

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.

4.1.1 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 }
 }
```

4.1.2 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.

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.

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.

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:

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

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) $d_0, .., d_i, .., d_n$.

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 [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.

4.1.3 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;
```

4.1.4 The parsing algorithm

The algorithm of the C5_scanf parser generator is an implementation of the Earley algorithm [8] 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 after 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.

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}
    }
  }
}
```

Arithmetic expressions

The next token declaration in Yacc:

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

describes the precedence and associativityi rules 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.

4.1.5 Semantic actions

Semantic actions in the Yacc style are implemented with the function C5_compil.

The nexr program shows the use of two action TIEs 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 2×3 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.

4.2 Related work

Most parsers in use today are based on efficient linear-time algorithms that accept a subset of CFGs (LL,LR or LALR) [21].

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 [10] is the first Earley parser generator along the lines of Yacc and DEEP [20] is an efficient directly-executable Earley parsing.

Finally, we did not found parser generators that accept grammars denoted with C types to produce transparent parsing as `C5_scanf` does.

4.3 Conclusions

The generic function `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.

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 transparent parsing. The programmer just need to define a type and the parsing result is an object of that type. Then, the resulting object can be *processed* using the `C5_compil`,`C5_lenSearch` and `C5_idxSearch` functions.

Chapter 5

Graphics programming in C5.

When a C library is presented, we usually expect the syntactical and semantical description of a set of functions.

In C5, we can introduce a library by describing the meaning of types related to a certain task. The most remarkable property of this kind of C5 libraries is that we can use the library by doing type declarations instead of function callings. This is an important change of the programming methodology: type declarations can now be a very expressive member of a program. Furthermore, as we will see later, a type declaration can be the main code of a program.

In this chapter [1], we start presenting the OPM machine, a small graphics library based on the art concepts of the Uruguayan painter Joaquín Torres García.

Finally, we show how the OPM machine is used by the generic function `opm_image_cons` to achieve a powerful page-description language.

5.1 Torres García's art conception.

The Oriented Port Machine (OPM)is a constructive graphic machine based on the color plane concept of the Uruguayan painter Joaquín Torres García (1874-1949).

At the same time, the graphic representation rules of the image constructor `opm_image_cons` were designed inspired by the *Constructive Universalism* of the Uruguayan painter.

Torres García has proposed an art conception that stands out for understand the constructive painting like a symbols structure. [25]

He produced an art movement based on two concepts:

[1]This chapter is based on the Master Thesis of Pablo Queirolo[28].

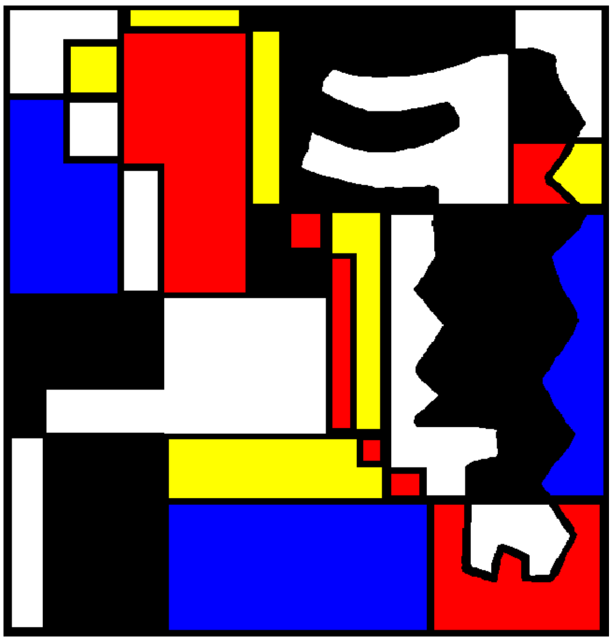

Figure 5.1: A Torres García's painting programmed by Pablo Queirolo.

Structure : in order to give a unity to the construction (*"Color planes and lines combined with art, will build a real structure."* [11]).

Abstraction : since he withdrew form imitation of nature, he defined ideograms to represent things and simple ideas in order to use universal representations (*"The painter is not interested in the object, he is interested in the color plane and the geometry of its structure."* [11]).

Torres García created a constructive painting based on a composition (structure) of rectangles (color planes) and ideograms.

A constructive imaging model, following Torres García ideas, can be presented in this way:

- construct the color planes of the page.
- construct a rectangle structure representing the image structure.
- for every ractangle of the structure, stamp an ideogram or construct a structure representing the rectangle image ...

  Continue this structuring process until the desired image is obtained.

This imaging conception –taking color rectangle structures as basic graphic objects– unifies the foreground-background duality; the classical duality of the painting model:

"*In the unity of the composition, the idea of thing and background should disappear. ... Then, there are not the thing and the background, all is thing and all is background.*"[11].

And when this duality disappear, the duality point-plane of the graphic machine disappear too, producing an important change: it is possible to design abstract graphic machines with independence of the pixel machine.

Torres García's concept of color plane has inspired *the oriented port machine*.

This concept of port is a generalization (unification) of the traditional concepts of port and pixel in Computer Graphics.

In Torres García's paintings we can find **structures**, **ideograms** and **color planes**. In C5, these concepts are implemented by type expressions with TIEs, DPTs and *oriented ports*.

5.2 The Oriented Port Machine

An oriented port is either a null port or a port representing a rectangular region of the *page*. The attributes of an oriented port are the coordinates, the color list and the orientation of the rectangular region. There are four different orientations: *Right*,*Down*, *Left* and *Up*. The current color of a non-null port is the first element of the color list. If the color list is null then the current color is *White*.

The *oriented port machine* is a C library based on the `Port_List` abstract data type:

- `Port_List opm_null()`
  The function returns a null port list.

- `Port_List opm_page(Color_List cl)`
  The function yields a one port list which is *Right* oriented, has the *page* size and the color list `cl` (Appendix A contains the defined colors of the Graphics Library).

- `Port_List opm_inters(Port_List pl1, Port_List pl2)`
  The function yields a port list constructed from the intersections (of the rectangular regions) of the cross product of the lists `pl1` and `pl2`.

  The color and orientation of the resulting ports are taken from the corresponding `lp1` ports. If the intersection is a line, a point or empty, then a null port is constructed.

- `Port_List opm_rot(int rot_nr,Port_List pl)`
  The function applies the rotation function to every port of the list `pl`.

  The ports are rotated `r` times according to the rotation rules for port orientation: $rotate(Right) = Down$, $rotate(Down) = Left$, $rotate(Left) = Up$ and $rotate(Up) = Right$.

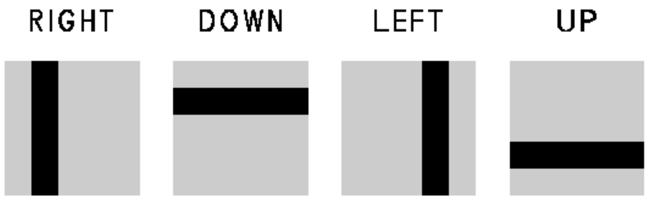

Figure 5.2: sel_split example for $n = 5$ and $i = 2$.

- `Port_List opm_selsplit(int n,int i,Port_List pl)`
  The function applies the function `sel_split` to every port of the list `pl`.

  If `n`>0 and `i` belongs to the range {`1`,`n`} then the function

  ```
  Port sel_split(int n,int i,Port p)
  ```

  splits the port `p` into `n` sub-rectangles and returns a port with the coordinates of the *ith* sub-rectangle. The orientation and color information are taken from `p`. Figure 5.2 shows the four different results of *sel_split* depending on the four possible orientations of `p`. If `i` is outside the range (`i`<0 or `n<i`) then the function returns a null port.

  Finally, in case of `n`≤ 0, the function returns a port equal to `p` for all value of `i`.

- `Port_List opm_partition(float f,Port_List pl)`
  The function applies the function `partition` to every port of the list `pl`.

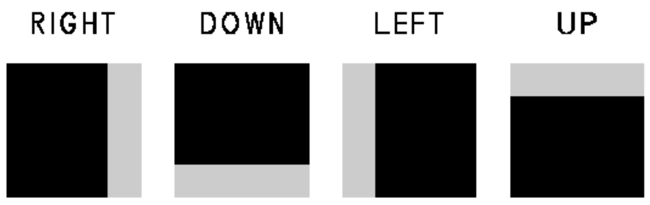

Figure 5.3: A partition example for $f = 0.75$

If $0 <$`f`< 1 then the function `Port partition(float f, Port p)` divides the port `p` into two rectangles proportionally to the floating number f and returns a port representing the first sub-rectangle.

The orientation and color information are taken from port `p`. Figure 5.3 shows the four different results depending on the four possible orientations of `p` (`f`$= 0.75$).

The function returns a null port if `f`≤ 0 and a port equal to `p` if `f`≥ 1.

- `Port_List opm_set_color(int n, Port_List pl)`
  The function applies the function `drop_color` to every port of the list `pl`.

  The function `Port drop_color(int n, Port p)` yields a port with the `p` color list discarding the first `n` elements if `n`> 0. The other attributes of the returned port are equal to those of `p`.

5.2.1 An *opm* example

Figure 5.4 shows a C program using the *opm* library. The function `opm_cat` concatenates two port lists and `opm_print` prints the graphic representation of the port list argument. The function `scale` is defined for scaling the letter `a` in the upper left and lower right corners.

The result of this example is an image (see Figure 5.5) with two black `a` letters in a gray background.

5.3 The C5 Graphics Library

Since a detailed description of the C5 Graphics Library is out of the scope of this chapter, we will concentrate our attention in the most important function

```
typedef struct{
              int y, x;
              } Ccoords;
typedef Ccoords Point_set[15];
Point_set points={
    { 6, 1 }, { 6, 2 }, { 5, 0 }, { 5, 3 },
    { 4, 3 }, { 3, 1 }, { 3, 2 }, { 3, 3 },
    { 2, 0 }, { 2, 3 }, { 1, 0 }, { 1, 3 },
    { 0, 1 }, { 0, 2 }, { 0, 4 }
    };
Port_List scale(double right, double down,
                double left, double up, Port_List lp){
    return(
        opm_inters(opm_partition(right,opm_rot(0,lp)),
        opm_inters(opm_partition(down ,opm_rot(1,lp)),
        opm_inters(opm_partition(left ,opm_rot(2,lp)),
             opm_partition(up   ,opm_rot(3,lp)))))
       );
    }
main(){
    int i;
    Port_List pl=opm_page(Gray85, Black, NULL), pl_2a;
    opm_print(pl);
    lp_2a=opm_cat(scale(0.8,0.8,0.4,0.4,opm_set_color(1,pl)),
                  scale(0.4,0.4,0.8,0.8,opm_set_color(1,pl))
                  );
    for(i=0;i<15;i++)
         opm_print(opm_inters(
            opm_selsplit(6+1,points[i].y+1,opm_rot(-1,pl_2a)),
            opm_selsplit(4+1,points[i].x+1,pl_2a))
            );
    }
```

Figure 5.4: A simple *opm* example

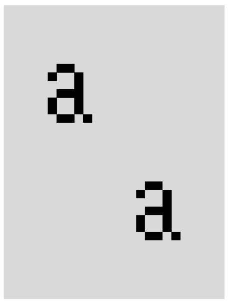

Figure 5.5: The letter `a` example.

of the library:

```
Port_List opm_image_cons(DPT dt, Port_List pl)
```

This function is an image constructor with a dependent pair argument. The semantics of the function `opm_image_cons` is informally explained by describing the graphic meaning of types with TIEs:

- Integer numbers: `DT_typedef int {m,n} Int_Def;`
  An `int` TIE is a two integers sequence `{m,n}` defining the visible range where `m` and `n` are the first and last visible integers respectively. If the dependent pair argument of `opm_image_cons` is `DT_pair(Int_Def,i)`, then the function returns the port list constructed by
  `opm_selsplit(n-m+1,i-m+1,pl)`.

- Floating point numbers: `DT_typedef float {s,t} Float_Def;`
  A `float` TIE is a two floating point numbers sequence `{s,t}` defining the visible range of the elements of this type. If the dependent pair argument of `opm_image_cons` is `DT_pair(Float_Def,f)` and `s<t`, then the function returns the port list constructed by
  `opm_partition((f-s)/(t-s),pl)`

  In case of `s`≥`t` , the function returns a port list equal to `pl`.

- Characters: `DT_typedef char {c1,c2,...,c8} Char_Def;`
  If `dt` is a pair with a `char` type definition including a TIE of eight floating point numbers, then the function `opm_image_cons` yields a port list representing the character font defined by the TIE values (see Appendix B for more information).

- Structures: `DT_typedef struct`$\{f_0, f_1, ..., f_n\}$`{r} Struct_Def;`
  A `struct` TIE is an integer `{r}` that defines the field rotation. The function `opm_image_cons` returns the port list generated by the intersection

of the graphic representation of the struct fields previously rotated $\texttt{r}\times i$ times ($0 \leq i \leq n$), where i is the index of the *ith* field of a structure of $n+1$ fields.

The intersections and rotations are implemented with `opm_inters` and `opm_rot` respectively.

The next C5 program is a short example using a structure of an integer and a floating point number:

```
DT_typedef int {0,2} Int_nr;
DT_typedef double {0.0,1.0} Double_nr;
DT_typedef struct{
                Int_nr n;
                Double_nr x;
                } {1} Struct_nx;
main(){
    Int_nr n=1;
    Double_nr x=0.6;
    Struct_nx nxs;
    nxs.n=n;
    nxs.x=x;
    opm_print(opm_image_cons(DT_pair(Int_nr,n),
        opm_page(Gray85,NULL)));
    opm_print(opm_image_cons(DT_pair(Double_nr,x),
        opm_rot(1,opm_page(Gray85,NULL)));
    opm_print(opm_image_cons(DT_pair(Struct_nx,nxs),
        opm_page(Black,NULL)));
    }
```

Notice that `Int_nr` and `Double_nr` are printed in gray while the struct *Struct_nx* is represented in black. This makes easier to see that the struct is the intersection of the representation of `n` and a $\Pi/2$ rotated `x` (see figure 5.6).

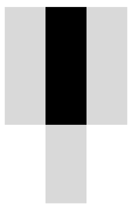

Figure 5.6: A simple struct declaration.

- Arrays: `DT_typedef Elems_type Array_Def[Max] {r,m,n};`
  An array TIE is a three integer sequence `{r,m,n}` where `r` defines the rotation of the elements of the array and `m` and `n` define the first and last visible array elements respectively.

  If `m` and `n` belong to the range `{0,Max-1}` then the elements of the array are represented according to the following rule: the *ith* element of the array is graphically represented on the port list constructed by

  ```
  opm_rot(r,opm_selsplit(n-m+1,i-m+1,pl))
  ```

- Unions: `DT_typedef union`$\{f_0, f_1, ..., f_n\}$`{c} Union_Def;`
  A `union` TIE is an integer `{c}` that defines the color of the fields. The function `opm_image_cons` returns the port list generated by the graphic representation of the valid union field with a current color defined by

  `opm_set_color(r`$\times$`i,pl)` $(0 \leq i \leq n)$

  where i is the index of the *ith* field of an union of $n + 1$ fields.

  Since C unions are not discriminated , the function `opm_image_cons` accepts a struct declaration with two fields, where the first is an union and the second an integer, like a discriminated union. In this case, the integer field is supposed to keep the information about the current field of the union. The discriminated union with TIE is the way to express in C5 the color structure of images.

- Pointers: `DT_typedef Ref_Obj * {r} Ptr_def;`
  A pointer TIE is an integer `{r}` that defines the rotation of the referenced object. The function `opm_image_cons` returns the graphic representation of the referenced object on a `r` times rotated `pl`.

- Type definitions: `DT_typedef Prev_Def {r,c} Def_Def;`
  Type definitions may include type declarations previously defined. In this case, the type definition TIE is a two integer sequence `{r,c}` where `r` and `c` set the rotation and current color of the defined type respectively.

A detailed version of the C5 Graphics Library including the specification of `opm_image_cons` can be found in Appendix C.

```
DT_typedef struct{
                 int {0,6} y;
                 int {0,4} x;
                 } Ccoords;
DT_typedef Ccoords {3,1} Point_set[15] {0,1,0};
Point_set pts={
    { 6, 1 }, { 6, 2 }, { 5, 0 }, { 5, 3 },
    { 4, 3 }, { 3, 1 }, { 3, 2 }, { 3, 3 },
    { 2, 0 }, { 2, 3 }, { 1, 0 }, { 1, 3 },
    { 0, 1 }, { 0, 2 }, { 0, 4 }
    };
DT_typedef struct{
          double {0.0,1.0} right, down, left, up;
          } Scale_2[2] {0,1,0};
Scale_2 scs={{0.8,0.8,0.4,0.4},{0.4,0.4,0.8,0.8}};
main(){
    Port_List lp=opm_page(Gray85, Black, NULL);
    opm_print(lp);
    opm_print(opm_image_cons(DT_pair(Point_set,pts),
            opm_image_cons(DT_pair(Scale_2,scs),lp)));
    }
```

Figure 5.7: A C5 version of the *opm* example.

5.3.1 A C5 version of the *opm* example

Figure 5.7 shows a C5 version of the *opm* example presented in figure 5.4.

The most relevant difference between both programs is that what the C5 program really do is mainly specified (programmed) in three `DT_typedef` declarations, while the rest of the program itself deals with the variables `pts` and `scs` construction and the image printing. Furthermore, the type definition `Scale_2` is enough expressive to substitute for the function `scale` of the C program.

5.4 Programming images in C5

The C5 Graphics Library transforms C5 in a high level page-description language, i.e., a language capable of describing the appearance of text, graphic shapes and images on a page. The use of DPTs and TIEs increase the abstraction level of the language producing a readable and compact code for graphics programs.

The image presented in figure 5.8 is generated by a 5 Kb C5 source program.

Some statistics will help us to show why this library produce such a compact code:

The program uses twelve graphics functions of the Standard Output Library in 103305 callings where 92 of them are invoked explicitly in the program and the others 103213 are called implicitly through 15 invocations of `opm_image_cons` which is the only Standard Output Library function with a DPT argument. Seven of these callings answer for the font construction involving 60% of the total quantity of callings.

Eight `DT_typedef` declarations were required by the 15 `opm_image_cons` invocations, remarking that types with TIEs are the heart of the design of programs that use this kind of libraries.

Finally, C5 translates this program into a 22 Kb C code which produces, when compiled and executed, a 3.1 Mb PostScript [3] file.

5.5 Related work

Constructive methodologies are not new in Computer Graphics. Constructive Solid Geometry (CSG) has been widely used in 3D Solid Modeling. The main idea in CSG is to describe a solid object as a composition of primitive objects (cylinders, spheres, cubes) combined with Boolean set operators

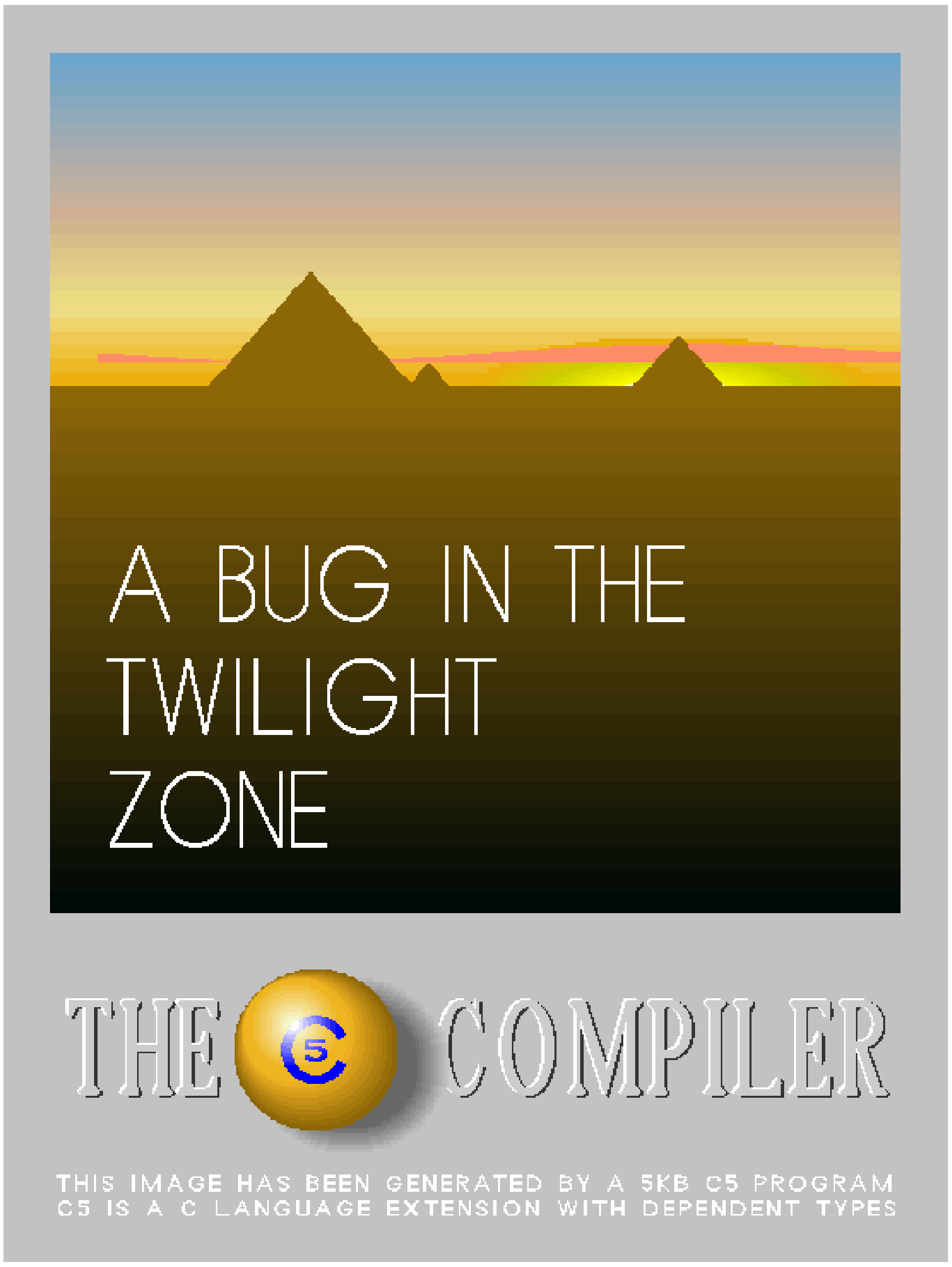
A BUG IN THE
TWILIGHT
ZONE
THE
5
COMPILER
THIS IMAGE HAS BEEN GENERATED BY A 5KB C5 PROGRAM
C5 IS A C LANGUAGE EXTENSION WITH DEPENDENT TYPES

Figure 5.8: An image programmed in C5

such as union, intersection and difference. An objet is stored as a tree with operators at the internal nodes and simple primitives at the leaves [14] [1] .

Although CSG is a simple and compact way of representing solids that induces to a constructive thinking when defining a 3D object, it is neither a constructive programming language nor a formal system with well-known properties.

There are also several approaches to express pictures with structured datatypes and functional programming producing low-cost prototypes that are easy-to-use for non expert graphics programmers [27] [9] [31].

The main difference with the C5 Graphics Library is that the graphics functions of these packages are based on the painting model. In other words, they define the line as a primutuve graphics function.

These approaches are an interesting innovation from the programming methodologies side but the lackness of a coherent imaging or page-description model reduced them to a friendly interface of other standard graphics packages or page-description languages like PostCript.

5.6 Conclusions

The results of the experimentation with the C5 Graphics Library indicates that:

- The Graohics Library of C5 is powerful enough to express complex images. The language do not require large function libraries to reach an acceptable expressiity level.
- The readability and reusability of the C5 Graphics Library is better than other graphics libraries of C based on the painting model.
- Most computing students who tested C5 and the Graphics Library consider that an important amount of time is required to learn the basic concepts of C5 and the constructive model of the Graphics Library.
- In the other hand, the students declare that the programming task was done in a short time and the resulting programs were very compact and easy to reuse.

Chapter 6

The C5 Graphics Tutorial

Let us begin with a quick introduction to the C5 Graphics Library. Our aim is to show the use of DPTs and TIEs in real programs, but without getting bogged down in details, formal rules or theoretical concepts.

6.1 Getting Started

The first program to write is the same for all languages:

Print the words HELLO WORLD

In C5, the program to print *HELLO WORLD* is

```
#define Max_Char 40
#define Str_TIE {0,0,Max_Char-1}
DT_typedef char Arial String[Max_Char] Str_TIE;
main(){
    String str="HELLO WORLD";
    opm_print(opm_image_cons(DT_pair(String,str),
        opm_page(Black,NULL)));
    }
```

and the resulting image is showed in figure 6.1. The type definition has two TIEs: `Arial` and `Str_TIE`. The first is attached to the `char` type specifying the font to be used when a character is printed and the second to the array type. The array TIE is a three integer expression with the following form:

{*rotation,first visible element,last visible element*}

where *rotation* is an integer specifying a clockwise rotation of the array elements with an angle of *rotation* $\times \Pi/2$, and the *first visible element* and *last*

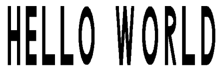

Figure 6.1: Hello World

visible element are integers defining the sequence of array elements that will be printed. In our program the TIE values {`0,0,Max_Char-1`} mean that the characters will be rotated 0 degree and all the array characters are visible for printing.

In the body of the main function, the function `opm_page` is a page constructor whose arguments are the colors (see apendix A) required by the image to be printed on that page. When the page is created , the current color is specified by the first argument (`Black` in our program). The type of the range of this function is `Port_List`. This is the type of the second argument of `opm_image_cons` –the *case-type* function of the library– and the argument of `opm_print` too. Notice that pages and images are represented by the same data structure.

6.2 Integer numbers

The C `printf` translates integer numbers to a digit character sequence starting with the minus character if the number is a negative integer.

Instead of this character oriented translation, the image constructor `opm_image_cons` represents integer numbers by simple geometric images based on color rectangles.

Let us see a short C5 program that prints the number 2 for a quick understanding of the way C5 produce the graphic representation of an integer:

```
DT_typedef int {0,3} intnr;
main(){
    intnr n=2;
    opm_print(opm_page(Gray85,NULL)); /* A gray background */
    opm_print(opm_image_cons(DT_pair(intnr,n),
                             opm_page(Black,NULL)));
    }
```

The `int` TIE is a two integer sequence with the following form:

{ *first visible integer* , *last visible integer* }

where the *first visible integer* and *last visible integer* are integers defining the printable range of numbers. The default TIE for the `int` type is {0,1}.

In our program the TIE values {`0,3`} mean that the integers 0, 1, 2 and 3 are visible for printing.

In this case, the page is virtually divided in four vertical rectangles. Starting from the left side of the page , the first rectangle represents the number 0, the second the number 1, the third the number 2 and the last rectangle the number 3. Accordingly, when printing the number 2, `opm_print` will print the third rectangle painted in black.

Figure 6.2 shows the resulting page.

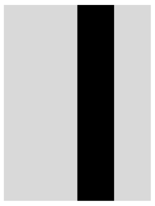

Figure 6.2: An integer number representation.

What would this program do if we try to print a number outside the range {0,3}? Suppose we have the number `11` for printing. Just the gray background will be printed because `11` is not a visible number in this range.

There is a way to express the range { $-\infty, +\infty$ } by declaring a `int` TIE of the form $\{m, n\}$ where $m > n$. For example, the TIE $\{1, 0\}$ specifies that all the integer numbers are visible. The graphic representation of an integer with infinite visibility is the complete page painted with the current color.

6.3 Floating point numbers

A visible floating point number is graphically represented by the first (left) rectangle of a proportional partition of the page.

A program that prints the number 2.5 shows how C5 produce the graphic representation of floating point numbers:

```
DT_typedef double {0.0,4.0} float_nr;
main(){
    float_nr f=2.5;
    opm_print(opm_page(Gray85,NULL)); /* A gray background */
    opm_print(opm_image_cons(DT_pair(float_nr,f),
        opm_page(Black,NULL)));
    }
```

The `double` or `float` type TIE is a two floating point number sequence with the following form:

$$\{ \textit{ first visible float , last visible float } \}$$

where the *first visible float* and *last visible float* are floating point numbers delimiting the printable range of numbers. The default TIE for `double` or `float` types is {0.0,1.0}.

In our program the TIE values `{0.0,4.0}` mean that a floating point number `f` is visible if `f`$\geq$ 0.0 and `f`$\leq$ 4.0. In this case, the page is partitioned in two rectangles with a `f` dependent size. The graphic representation of `f` is the left rectangle painted with the current color. If $f > 4.0$ the graphic representation will be the complete page painted with the current color and if $f < 0.0$ no action is produced and the resulting image is the page painted with the background color.

Figure 6.3 shows the resulting page.

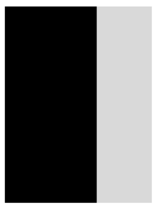

Figure 6.3: A floating point number representation.

6.4 Structures

The type `struct` is represented by the intersection of its fields rotated with an angle determined by the field index and the struct TIE.

Let be the following struct declaration of $n+1$ fields:

$$\textit{DT\_typedef struct}\{\ d_0, .., d_i, .., d_n\ \}\ \{r\}\ \textit{sd;}$$

where $d_0, .., d_i, .., d_n$ are C5 type declarations. The graphic representation for the struct type is the intersection of the rotated graphic representation of $d_0, .., d_i, .., d_n$. The rotation angle for the i^{th} field of the struct sd is

$$r \times i \times \Pi/2$$

where r is the rotation declared in the struct TIE. The default struct TIE is 1. The struct representation will be painted with the current color of the first field.

The next program shows how the graphic representation of the type `struct` is generated by `opm_image_cons`:

```
DT_typedef int {0,2} int_nr;
DT_typedef double {0.0,1.0} double_nr;
DT_typedef struct{
                int_nr n;
                double_nr x;
                } {1} nx_Struct;
main(){
    int_nr n=1;
    double_nr x=0.6;
    nx_Struct nxs;
    nxs.n=n;
    nxs.x=x;
    opm_print(opm_image_cons(DT_pair(int_nr,n),
        opm_page(Gray85,NULL)));
    opm_print(opm_image_cons(DT_pair(double_nr,x),
        opm_rot(1,opm_page(Gray85,NULL)));
    opm_print(opm_image_cons(DT_pair(Struct_nx,nxs),
        opm_page(Black,NULL)));
    }
```

Notice that `int_nr` and `double_nr` are printed in gray while the struct *Struct_nx* is represented in black. This makes easier to see that the struct is the intersection of the representation of `n` and a $\Pi/2$ rotated `x` (see figure 6.4).

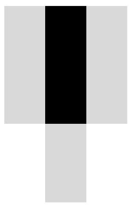

Figure 6.4: A simple struct declaration.

The struct type with TIEs is an expressive programming resource of the C5 Graphics Library. The code of `opm_scale` illustrates how a member of this library has been programmed:

```
DT_typedef struct{
            double {0.0,1.0} x2,y1,x1,y2;
            } {1} dp2;

Port_List opm_scale(double left, double right,
                    double up,   double down,
                    Port_List pl){
        dp2  margs;
        margs.x1= left ;
        margs.x2= right;
        margs.y2= up;
        margs.y1= down;
        if(pl==NULL) return(NULL);
        else return(opm_cat(
                opm_image_cons(DT_pair(dp2,margs),
                        opm_cons(opm_hd(pl),NULL)),
                opm_scale(left,right,up,down,opm_tl(pl))));
        }
```

The function `opm_cat` is the concatenation operator for port lists and the functions `opm_hd` and `opm_tl` return the head and the tail of a port list respectively.

Let us look closer at the struct declaration because there are two interesting things to note here. First, since the rectangles representing the variables $x2, y1, x1$ and $y2$ are rotated 0, 1, 2 and 3 times $\Pi/2$ respectively, the intersection produced by the struct `dp2` will be a scaled rectangle defined by the values of the `x2,y1,x1` and `y2` variables. Second, what the function `opm_scale` really do is mainly specified (programmed) in the struct declaration while the body of the function itself deals with the `margs` variable assigning and the port list `pl` recursive handling.

6.5 Arrays

In the next program, the function *sin* is visualized using an array of floating point numbers. This example is interesting because it shows the graphic power of this type for function visualization:

```
#define Max 100
DT_typedef double {-1.0,1.0} func_visual[Max] {3,0,Max-1};
main(){
      func_visual fn;
      double rn=0.0;
      int i;
```

```
        for(i=0;i<Max;i++){
              fn[i]= sin(rn);
              rn= rn + 2.0*M_PI/Max;  /* M_PI=3.1416... */
              }
        opm_print(opm_page(Gray85,NULL)); /* A gray background */
        opm_print(opm_image_cons(DT_pair(func_visual,fn),
              opm_page(Black,NULL)));
        }
```

The array TIE specifies that all the array elements are visible and will be rotated $3 \times \Pi/2$ before printing. The array `fn` is assigned with `sin` values in the range $\{0, 2\Pi\}$ and then visualized.

Figure 6.5 shows the *sin* function visualization.

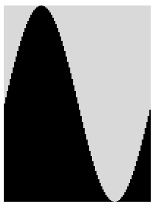

Figure 6.5: The sin function visualization.

6.5.1 The Set mode

As we did for the integer TIE, it is possible to define an infinite range TIE for arrays.

We call the *Set Mode* representation to an array declaration with TIEs of the form

$$\{ \; rot, \; m \; ,n \; \}$$

where *rot* specifies the rotation of the array elements and m and n are integers so that $m > n$.

In this case , the elements of the array are represented directly on the page following the order indexed by the array starting from 0.

The next program shows the *Set Mode* representation in an array declaration of integer structures:

```
  DT_typedef struct{
              int {0,6} y;
              int {0,4} x;
              } ccoords;
```

```
DT_typedef ccoords point_set[15] {0,1,0};

point_set pts={
    { 6, 1}, { 6, 2}, { 5, 0}, { 5, 3}, { 4, 3},
    { 3, 1}, { 3, 2}, { 3, 3}, { 2, 0}, { 2, 3},
    { 1, 0}, { 1, 3}, { 0, 1}, { 0, 2}, { 0, 4} };

main(){
    opm_print(opm_page(Gray85,NULL));
    opm_print(opm_image_cons(DT_pair(point_set,pts),
        opm_rot(3,opm_page(Black,NULL)));
    }
```

This is an important array declaration because it simulates a two dimensional Cartesian Coordinate System. In the program, the points are pairs of integers where the first element is the Y axis coordinate and the second, the X axis. Notice again that the kernel of the program is the type declaration.

The resulting image is a 15 points Cartesian representation of the letter a.

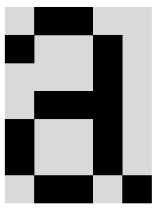

Figure 6.6: The Set Mode Representation.

6.5.2 Matrices.

Matrix representation is obtained in C5 by the double array declaration with finite range TIEs. The following program produce the same output than the previous but now the letter a is represented by a 7×5 matrix:

```
DT_typedef int {1,1} matrix[5] {0,0,4} [7] {3,0,6};
matrix mtx={  0,1,1,0,0,
              1,0,0,1,0,
              0,0,0,1,0,
              0,1,1,1,0,
              1,0,0,1,0,
              1,0,0,1,0,
              0,1,1,0,1 };
```

```
main(){
    opm_print(opm_page(Gray85,NULL));
    opm_print(opm_image_cons(DT_pair(matrix,mtx),
        opm_rot(1,opm_page(Black,NULL))));
    }
```

There are some interesting details here. First, since the integer TIE range is $\{1,1\}$, only one number –the number one– is visible for printing. Second, the way the double array `mtx` is initialized makes easy the graphic design of the matrix. Third, the port list `opm_page(Black,NULL)` is rotated by `opm_rot` so that the printed matrix (in this case , the letter a) is coincident with the `mtx` initialization.

6.6 Unions

C Unions are not interesting for C5 programs because there is no way to know the current field of an union.

Instead of this kind of union , C5 recognizes discriminated unions as a special case of the struct type.

6.6.1 Discriminated unions

A struct declaration with two fields where the first is an union and the second is an integer is recognized as a discriminated union. In this case, the integer field is supposed to keep the information about the current field of the union. The discriminated union with TIEs is the way we express the color structure of the images printed by `opm_printf`.

The graphic representation of the discriminated union follows the struct rules and the representation of the union field is the representation of the current field painted with a color determined by the union TIE and the place of the field.

The form of a discriminated union is:

$$\begin{array}{ll} \mathit{DT\_typedef} & \mathit{struct}\{ \\ & \mathit{union}\{\ d_0,..,d_i,..,d_n\ \}\ \{c\}\ \ <\mathit{id}>; \\ & \mathit{int}\ \{m,n\}\ \ <\mathit{id}>; \\ & \}\ \{\ r\ \}\ \ <\mathit{id}>; \end{array}$$

where $d_0,..,d_i,..,d_n$ are the fields of the union, and c is an integer number defining the color factor of the union.

The current color of the *ith* field of the union is the $(c \times i + 1)^{th}$ element of the color list of the page.

The next program shows a discriminated union example:

```
DT_typedef struct{
              union{
                  int {0,9} foreground;
                  double {0.0,1.0} background;
                  } {2} cu; /* the color factor is 2 */
              int {1,0} idx;
              } disc_union ;
main(){
    Port_List pl=opm_page(Black,Red,Gray85,NULL);
    disc_union du1,du2;
    du1.idx=1;
    du1.cu.background=0.75; /* Gray85 */
    du2.idx=0;
    du2.cu.foreground=4;    /* Black  */
    opm_print(opm_image_cons(DT_pair(disc_union,du1),pl));
    opm_print(opm_image_cons(DT_pair(disc_union,du2),pl));
    }
```

The variable `foreground` –the first field of the union `cu`– is printed with the color `Black` because the equation $field_place \times color_factor + 1$ is $0 \times 2 + 1$ and this implies that the current color is the first color of the page. In the case of the variable `background`, the resulting equation is $1 \times 2 + 1$, that is, the third color of the page (`Gray85`).

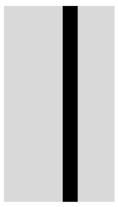

Figure 6.7: A discriminated union representation.

The output of `opm_print` is presented in Figure 6.7.

6.7 Pointers and recursion

The rule for the graphic representation of a pointer type declaration is the representation of the pointed object with a rotation specified by the pointer

TIE. The default TIE for pointers is {0}. If the referenced object is a `char`, C5 will try to represent a NULL terminated string likewise an array of characters.

The form of a pointer TIE is { *rotation* } where *rotation* is an integer.

In the C language, recursive type declarations are expressed by a struct declaration including a field with a pointer to itself. This kind of recursive declarations is required when implementing lists, trees or any other dynamic data structure.

When C5 recognizes a recursive struct declaration, it represents the objects of this type in Set Mode, i.e., the fields of the struct are printed in an ordered sequence, discarding the intersection operator that is applied in non recursive struct declarations.

The program below shows a recursive type declaration for implementing a list of points:

```
DT_typedef struct{
              double {-500.0,10000.0} y;
              int {-100,100} x;
              } point;
DT_typedef struct NODE{
              point pt;
              struct NODE *next;
              } {0} *node_list;
DT_typedef struct NODE * {3} Node_List;

node_list ucons(double y, int x, node_list l){
    node_list p;
    p= (node_list) malloc(sizeof(struct NODE));
    p->pt.y=y;
    p->pt.x=x;
    p->next=l;
    return(p);
    }

main(){
    Node_List nl=NULL;
    int x;
    for(x=-100;x<=100;x++) nl=ucons((double) x*x,x,nl);
    opm_print(opm_page(Gray55,NULL));
    opm_print(opm_image_cons(DT_pair(Node_List,nl),
        opm_page(Black,NULL)));
    }
```

The `point` struct produce the intersection of `y` and `x` while the struct `NODE` is recursive and therefore it generates the image of the fields without intersections.

In order to keep respectively the `y` and `x` coordinates vertical and horizontal, the pointer `Node_List` is declared including a TIE that specifies a rotation $3 \times \Pi/2$.

The visualization of the function $f(x) = x^2$ is predented in Figure 6.8.

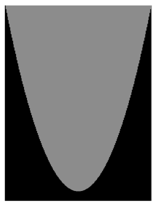

Figure 6.8: A recursive type representation.

6.7.1 Color expressions

Dynamic data structures like lists or binary trees with nodes including unions are natural implementations for color expressions in C5. As a good example, let us write the type declaration of `Color_Serie` which is the output type of the function `opm_colors`, the color expression constructor of the C5 Graphics Library.

```
DT_typedef struct OPMTON{
              dp2 scale;
              struct{ /* discr union  */
                      union{
                             int bg;
                             struct OPMTON *next;
                             } un;
                      int{1,0} idx;  /* infinite range */
                      } du;
              }{0} *Color_Serie;
```

The next program shows how color tones are structured with text using `opm_colors`:

```
DT_typedef struct{
              Color_Serie * {3} c;
              char Antique_Draft_S string[20] {0,0,19};
              } {0} Color_String;
main(){
```

```
Port_List lp1,lp2;
Color_String cst;
Color_Serie obj=opm_colors(4,2*TONES_NR,1.0,0.0);
cst.c=&obj;
strcpy(cst.string,"AB");
lp1=opm_page(opm_col2col( White,  Gray50,    TONES_NR),
             opm_col2col(Gray50,   White,    TONES_NR),
             NULL);
lp2=opm_page(opm_col2col( Black,   White,    TONES_NR),
             opm_col2col( White,   Black,    TONES_NR),
             NULL);
opm_print(opm_image_cons(DT_pair(Color_Serie,obj),
        opm_scale(0.75,0.75,0.90,0.90,opm_rot(1,lp1))));
opm_printf(opm_image_cons(DT_pair(Color_String,cst),lp2));
}
```

The function `opm_col2col` is a compressed notation for color series. For example, the color serie denoted by

`opm_col2col(White,Black, 4)`

is, when expanded, equivalent to the color series

`(White,Gray67,Gray33,Black)`

.

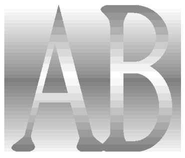

Figure 6.9: Color tones and text.

Figure 6.9 shows the output of `opm_print` .

6.8 Type definitions and enumerations

Type definitions may include type declarations previously defined. In this case, the TIE is of the form:

$$\{rotation, color\}$$

where *rotation* and *color* are integers that work similar to the struct and union TIE respectively. The default TIE for type definitions is $\{0, 0\}$.

The rule for representing enumerations is quite simple. Let be the type declaration

$$DT_typedef enum\{id_0, id_1, ..., id_n\}enum\_id;$$

where id_i are $n+1$ non equal identifiers. The objects of this type will be represented in a similar way to the type declaration

```
DT_typedef int {0,n} enum_index;
```

Chapter 7

About the C5 compiler.

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

In this chapter, we present the version 0.98 of the C5 compiler (September, 2006), some references to related work and the conclusions of the experimentation with C5.

7.1 The version 0.98 of the C5 compiler

The C5 parser is a extended C parser with few grammatical modifications (the syntax of C5 is presented in Appendix E). 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 mainly concerned with the constructor `DT_pair`:

$$DT\_pair(\ A\ ,\ a\ )$$

where A is a C5 type definition

$$DT\_typedef \ < type\_expr > \ A \ ;$$

and a is a variable of type A.

When a `DT_pair` invocation is detected, the C5 compiler checks statically if the first argument is a `DT_typedef` type definition and if the second is a variable of type equal to the value of the first argument.

The current implementation of the C5 compiler (Version 0.98, September 2006) is available for the Linux operating system.

The C5 compiler and a sample of C5 programs can be found on the Web at

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

7.2 Related work

The statically typed programming languages Amber [7] and Modula-3 [26] include notions of a dynamic type and a typecase statement. This idea can also be found in functional programming [23] [29] [18] and in type-safe C dialects like Ccured [12] 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 [4] –a Haskell-like [17] 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 [6][15]. is developing another approach. PolyP [19] 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.

7.3 Conclusions

Generic programming is a complex task. C5 has been designed to be a low cost C extension to obtain a generic programming framework.

The practice with C5 allows us to affirm that C5 is a useful generic programming framework.

The most important conclusion of the experimentation with C5 is that a static typed language extended with DPTs (dynamics) and TIEs can be powerful enough to express a wide class of generic functions (i.e. `C5_scanf`) in a straightforward, compact and safe way.

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

TIEs seem to be a friendly way of providing parameters for generic functions without affecting the static C type system. In the case of `C5_compil`, the use of TIEs with C5 functions allows C5 to express a generic function in a very compact and readable 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 complex generic functions like `C5_scanf`.

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.

Finally, we like to remark that the results of the C5 experimentation is not limited to the C language. DPTs and TIEs are generic concepts and they can be applied in other static typed programming languages. For instance, a new version of generic Haskell inspired on the C5 ideas was developed at InCo in 2006 [30].

Chapter 8

Epilog

The main goal of the C5 project is to experiment with programming languages that support static and dynamic typing in coexistence mode. Thus, the way the static and dynamic universes are communicated is a critical point of the programming behavior of such languages.

The practice with C5 showed that the way C5 communicates between the static and the dynamic universes provides the expressiveness required to program generic functions defined for the entire type system.

This is an important empirical result since the communication from the dynamic to the static universe in C5 is restricted to atomic types, and this is a strong restriction compared with Cardelli's typecase statement.

The experimentation with C5 convinced us that `C5_gos` and `C5_fapply` give to C5 an equivalent expressiveness than Cardelli's typecase statement.

However, what the practice really showed was the importance of the TIEs when programming complex programs like the parser generator or the image constructor.

In such cases, generic functions cannot be properly defined using just the –one dimension– function arguments.

The functions `C5_compil`, `C5_scanf` or `opm_image_cons` are relevant examples to see how useful are TIEs for generic programming.

DPTs are necessary for generic programs to answer the question

What is the type of this object?

but TIEs are required to answer an even more important question

What do you want to do with this object?

The power of generic functions has an obvious counterpart: programming generic functions is a complex task. The practice with C5 confirmed this assertion. The design, implementation and testing of functions like `C5_scanf` or `opm_image_cons` required a considerable effort.

In the other hand, the practice showed also that the use of generic functions do not present special difficulties.

For instance, about 400 undergraduate computing students at InCo showed that they can use C5 libraries (including generic functions with DPTs and TIEs) to construct small C5 programs (100-300 lines) as good as they do with other standard programming languages like C. C++ or Java. However, 60% of the students declare that they required –in comparison with C++ or Java– additional time to understand properly the new concepts of C5.

In the case of graduate students with C language and functional programming knowledge, we have not detected specific difficulties for generic programming in C5.

Finally, we like to present some statistics about the C5 project in order to give a global vision of its significance.

C5 statistics

program	prog. languages	program size (lines)	man-hours
C5 compiler	C YACC LEX	5500	1800
DPT Library	C5	1600	1200
Graphics Library	C5	3400	1400
C5_scanf	C5	2200	950

Bibliography

[1] G. Wyvill and T. L. Kunii and Y. Shirai. Space Division for Ray Tracing in CSG. *Computer Graphics & Applications*, 6:28–34, April 1986.

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

[3] Adobe. *Postcript Language Reference Manual.* Adisson Wesley, second edition, 1990.

[4] 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.

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

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

[7] 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. Also AT&T Bell Laboratories Technical Memorandum TM 11271-840924-10.

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

[9] Emmanuel Chailloux and Guy Cousineau. Programming Images in ML. In *ACM SIGPLAN workshop on ML and its aplications.*, 1992.

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

[11] Joaquín Torres García. *Lo aparente y lo concreto en el arte (1947).* Centro Editor de América Latina, capítulo oriental 41 edition, 1969.

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

[13] GNU. *Using and Porting the GNU Compiler Collection (GCC).* Free Software Foundation, http://gcc.gnu.org/onlinedocs/gcc-2.95.3/gcc.html, 2006.

[14] Goldstein R. A. and Nagel R. 3D Visual Simulation. *Simulation*, pages 25–31, January 1971.

[15] 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.

[16] Allen I. Holub. *Compiler Design in C.* Prentice Hall, appendix d a c grammar edition, 1990.

[17] 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.

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

[19] 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.

[20] John Aycock and Nigel Horspool. Directly-Executable Earley Parsing. *Lecture Notes in Computer Science*, 2027:229+, 2001.

[21] 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.

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

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

[24] 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.

[25] Gabriel Paluffo Linari. *Historia de la Pintura Uruguaya. Torres García : de Barcelona a París*. Ediciones de la Banda Oriental, 1992.

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

[27] Peter Henderson. Functional Geometry. In *ACM symposium on LISP and functional programming.*, pages 179–187, 1982. ISBN 0-89791-082-6.

[28] Pablo Queirolo. Typed Windos: An Implementation of a Programming Language for Graphic Design. Technical Report Master Thesis - 97-02, InCo PEDECIBA-Informática, 1997.

[29] 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.

[30] Adrian Sieradzki. Tipos dinámicos en lenguajes funcionales. Technical report, InCo PEDECIBA-Informática, 2006.

[31] Sigbjorn Finne and Simon L. Peyton Jones. Picture: A Simple Structured Graphics Model. In *Functional Programming*, page 4, 1995.

[32] 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.

Appendix A

Defined colors in the C5 Graphics Library.

AliceBlue
AntiqueWhite
AntiqueWhite1
AntiqueWhite2
AntiqueWhite3
AntiqueWhite4
Aquamarine
Aquamarine1
Aquamarine2
Aquamarine3
Aquamarine4
Azure
Azure1
Azure2
Azure3
Azure4
Beige
Bisque
Bisque1
Bisque2
Bisque3
Bisque4
Black
BlanchedAlmond
Blue
Blue1
Blue2
Blue3
Blue4
BlueViolet
Brown
Brown1
Brown2
Brown3
Brown4
Burlywood
Burlywood1
Burlywood2
Burlywood3
Burlywood4
CadetBlue
CadetBlue1
CadetBlue2
CadetBlue3
CadetBlue4
Chartreuse
Chartreuse1
Chartreuse2
Chartreuse3
Chartreuse4
Chocolate
Chocolate1
Chocolate2

```
Chocolate3
Chocolate4
Coral
Coral1
Coral2
Coral3
Coral4
CornflowerBlue
Cornsilk
Cornsilk1
Cornsilk2
Cornsilk3
Cornsilk4
Cyan
Cyan1
Cyan2
Cyan3
Cyan4
DarkGoldenrod
DarkGoldenrod1
DarkGoldenrod2
DarkGoldenrod3
DarkGoldenrod4
DarkGreen
DarkKhaki
DarkOliveGreen
DarkOliveGreen1
DarkOliveGreen2
DarkOliveGreen3
DarkOliveGreen4
DarkOrange
DarkOrange1
DarkOrange2
DarkOrange3
DarkOrange4
DarkOrchid
DarkOrchid1
DarkOrchid2
DarkOrchid3
DarkOrchid4
DarkSalmon
DarkSeaGreen
DarkSeaGreen1
DarkSeaGreen2
DarkSeaGreen3
DarkSeaGreen4
DarkSlateBlue
DarkSlateGray
DarkSlateGray1
DarkSlateGray2
DarkSlateGray3
DarkSlateGray4
DarkSlateGrey
DarkTurquoise
DarkViolet
DeepPink
DeepPink1
DeepPink2
DeepPink3
DeepPink4
DeepSkyBlue
DeepSkyBlue1
DeepSkyBlue2
DeepSkyBlue3
DeepSkyBlue4
DimGray
DimGrey
DodgerBlue
DodgerBlue1
DodgerBlue2
DodgerBlue3
DodgerBlue4
Firebrick
Firebrick1
Firebrick2
Firebrick3
Firebrick4
FloralWhite
ForestGreen
Gainsboro
GhostWhite
Gold
```

```
Gold1
Gold2
Gold3
Gold4
Goldenrod
Goldenrod1
Goldenrod2
Goldenrod3
Goldenrod4
Gray
Gray0
Gray1
Gray10
Gray100
Gray11
Gray12
Gray13
Gray14
Gray15
Gray16
Gray17
Gray18
Gray19
Gray2
Gray20
Gray21
Gray22
Gray23
Gray24
Gray25
Gray26
Gray27
Gray28
Gray29
Gray3
Gray30
Gray31
Gray32
Gray33
Gray34
Gray35
Gray36
Gray37
Gray38
Gray39
Gray4
Gray40
Gray41
Gray42
Gray43
Gray44
Gray45
Gray46
Gray47
Gray48
Gray49
Gray5
Gray50
Gray51
Gray52
Gray53
Gray54
Gray55
Gray56
Gray57
Gray58
Gray59
Gray6
Gray60
Gray61
Gray62
Gray63
Gray64
Gray65
Gray66
Gray67
Gray68
Gray69
Gray7
Gray70
Gray71
Gray72
```

Gray73
Gray74
Gray75
Gray76
Gray77
Gray78
Gray79
Gray8
Gray80
Gray81
Gray82
Gray83
Gray84
Gray85
Gray86
Gray87
Gray88
Gray89
Gray9
Gray90
Gray91
Gray92
Gray93
Gray94
Gray95
Gray96
Gray97
Gray98
Gray99
Green
Green1
Green2
Green3
Green4
GreenYellow
Grey
Grey0
Grey1
Grey10
Grey100
Grey11
Grey12
Grey13
Grey14
Grey15
Grey16
Grey17
Grey18
Grey19
Grey2
Grey20
Grey21
Grey22
Grey23
Grey24
Grey25
Grey26
Grey27
Grey28
Grey29
Grey3
Grey30
Grey31
Grey32
Grey33
Grey34
Grey35
Grey36
Grey37
Grey38
Grey39
Grey4
Grey40
Grey41
Grey42
Grey43
Grey44
Grey45
Grey46
Grey47
Grey48
Grey49

```
Grey5
Grey50
Grey51
Grey52
Grey53
Grey54
Grey55
Grey56
Grey57
Grey58
Grey59
Grey6
Grey60
Grey61
Grey62
Grey63
Grey64
Grey65
Grey66
Grey67
Grey68
Grey69
Grey7
Grey70
Grey71
Grey72
Grey73
Grey74
Grey75
Grey76
Grey77
Grey78
Grey79
Grey8
Grey80
Grey81
Grey82
Grey83
Grey84
Grey85
Grey86
Grey87
Grey88
Grey89
Grey9
Grey90
Grey91
Grey92
Grey93
Grey94
Grey95
Grey96
Grey97
Grey98
Grey99
Honeydew
Honeydew1
Honeydew2
Honeydew3
Honeydew4
HotPink
HotPink1
HotPink2
HotPink3
HotPink4
IndianRed
IndianRed1
IndianRed2
IndianRed3
IndianRed4
Indianred
Ivory
Ivory1
Ivory2
Ivory3
Ivory4
Khaki
Khaki1
Khaki2
Khaki3
Khaki4
Lavender
```

LavenderBlush
LavenderBlush1
LavenderBlush2
LavenderBlush3
LavenderBlush4
LawnGreen
LemonChiffon
LemonChiffon1
LemonChiffon2
LemonChiffon3
LemonChiffon4
LightBlue
LightBlue1
LightBlue2
LightBlue3
LightBlue4
LightCoral
LightCyan
LightCyan1
LightCyan2
LightCyan3
LightCyan4
LightGold
LightGoldenrod
LightGoldenrod1
LightGoldenrod2
LightGoldenrod3
LightGoldenrod4
LightGoldenrodYellow
LightGray
LightGrey
LightPink
LightPink1
LightPink2
LightPink3
LightPink4
LightSalmon
LightSalmon1
LightSalmon2
LightSalmon3
LightSalmon4
LightSeaGreen
LightSkyBlue
LightSkyBlue1
LightSkyBlue2
LightSkyBlue3
LightSkyBlue4
LightSlateBlue
LightSlateGray
LightSlateGrey
LightSteelBlue
LightSteelBlue1
LightSteelBlue2
LightSteelBlue3
LightSteelBlue4
LightYellow
LightYellow1
LightYellow2
LightYellow3
LightYellow4
LimeGreen
Linen
Magenta
Magenta1
Magenta2
Magenta3
Magenta4
Maroon
Maroon1
Maroon2
Maroon3
Maroon4
MediumAquamarine
MediumBlue
MediumOrchid
MediumOrchid1
MediumOrchid2
MediumOrchid3
MediumOrchid4
MediumPurple
MediumPurple1
MediumPurple2

```
MediumPurple3
MediumPurple4
MediumSeaGreen
MediumSlateBlue
MediumSpringGreen
MediumTurquoise
MediumVioletRed
MidnightBlue
MintCream
MistyRose
MistyRose1
MistyRose2
MistyRose3
MistyRose4
Moccasin
NavajoWhite
NavajoWhite1
NavajoWhite2
NavajoWhite3
NavajoWhite4
Navy
NavyBlue
OldLace
OliveDrab
OliveDrab1
OliveDrab2
OliveDrab3
OliveDrab4
Orange
Orange1
Orange2
Orange3
Orange4
OrangeRed
OrangeRed1
OrangeRed2
OrangeRed3
OrangeRed4
Orangered
Orchid
Orchid1
Orchid2
Orchid3
Orchid4
PaleGoldenrod
PaleGreen
PaleGreen1
PaleGreen2
PaleGreen3
PaleGreen4
PaleTurquoise
PaleTurquoise1
PaleTurquoise2
PaleTurquoise3
PaleTurquoise4
PaleVioletRed
PaleVioletRed1
PaleVioletRed2
PaleVioletRed3
PaleVioletRed4
PapayaWhip
PeachPuff
PeachPuff1
PeachPuff2
PeachPuff3
PeachPuff4
Peru
Pink
Pink1
Pink2
Pink3
Pink4
Plum
Plum1
Plum2
Plum3
Plum4
PowderBlue
Purple
Purple1
Purple2
Purple3
```

Purple4
Red
Red1
Red2
Red3
Red4
RosyBrown
RosyBrown1
RosyBrown2
RosyBrown3
RosyBrown4
RoyalBlue
RoyalBlue1
RoyalBlue2
RoyalBlue3
RoyalBlue4
SaddleBrown
Saddlebrown
Salmon
Salmon1
Salmon2
Salmon3
Salmon4
SandyBrown
SeaGreen
SeaGreen1
SeaGreen2
SeaGreen3
SeaGreen4
Seashell
Seashell1
Seashell2
Seashell3
Seashell4
Sienna
Sienna1
Sienna2
Sienna3
Sienna4
SkyBlue
SkyBlue1
SkyBlue2
SkyBlue3
SkyBlue4
SlateBlue
SlateBlue1
SlateBlue2
SlateBlue3
SlateBlue4
SlateGray
SlateGray1
SlateGray2
SlateGray3
SlateGray4
SlateGrey
Snow
Snow1
Snow2
Snow3
Snow4
SpringGreen
SpringGreen1
SpringGreen2
SpringGreen3
SpringGreen4
SteelBlue
SteelBlue1
SteelBlue2
SteelBlue3
SteelBlue4
Tan
Tan1
Tan2
Tan3
Tan4
Thistle
Thistle1
Thistle2
Thistle3
Thistle4
Tomato
Tomato1

```
Tomato2
Tomato3
Tomato4
Turquoise
Turquoise1
Turquoise2
Turquoise3
Turquoise4
Violet
VioletRed
VioletRed1
VioletRed2
VioletRed3
VioletRed4
Wheat
Wheat1
Wheat2
Wheat3
Wheat4
White
WhiteSmoke
Yellow
Yellow1
Yellow2
Yellow3
Yellow4
YellowGreen
```

Appendix B

Fonts of the C5 Graphics Library

In C5, fonts are represented by port lists. Fonts are the images associated to the `char` type. The `char` type TIE selects the desired font in a C5 program. There is a small set of predefined fonts (TIEs).

The rules for font names are

```
<Font>_<Draft|Letter|Ultra>_<UC|C|S|D|UD>
```

where `Font` is the font name (for example `Roman`), `Draft`, `Letter` and `Ultra` is the font resolution (quality) and the separation between fonts is represented by `UC` ultracompact, `C` compact, `S` standard, `D` disperse and `UD` ultra disperse.

The following is the form of a `char` type TIE:

```
{recnr,ftype,vert1,vert2,hor,serif,incl,disp}
```

B.1 A list of font TIES

```
#define Roman          {-1.0,1.0,0.1,0.4,0.15,0.2,0.0,0.7 }
#define Roman_Draft_C  {-1.0,1.0,0.1,0.4,0.15,0.2,0.0,0.7 }
#define Roman_Letter_C {-2.0,1.0,0.1,0.4,0.15,0.2,0.0,0.7 }
#define Roman_Ultra_C  {-3.0,1.0,0.1,0.4,0.15,0.2,0.0,0.7 }

#define Roman_Draft_UC  {-1.0,1.0,0.1,0.4,0.15,0.2,0.0,0.8 }
#define Roman_Letter_UC {-2.0,1.0,0.1,0.4,0.15,0.2,0.0,0.8 }
#define Roman_Ultra_UC  {-3.0,1.0,0.1,0.4,0.15,0.2,0.0,0.8 }
```

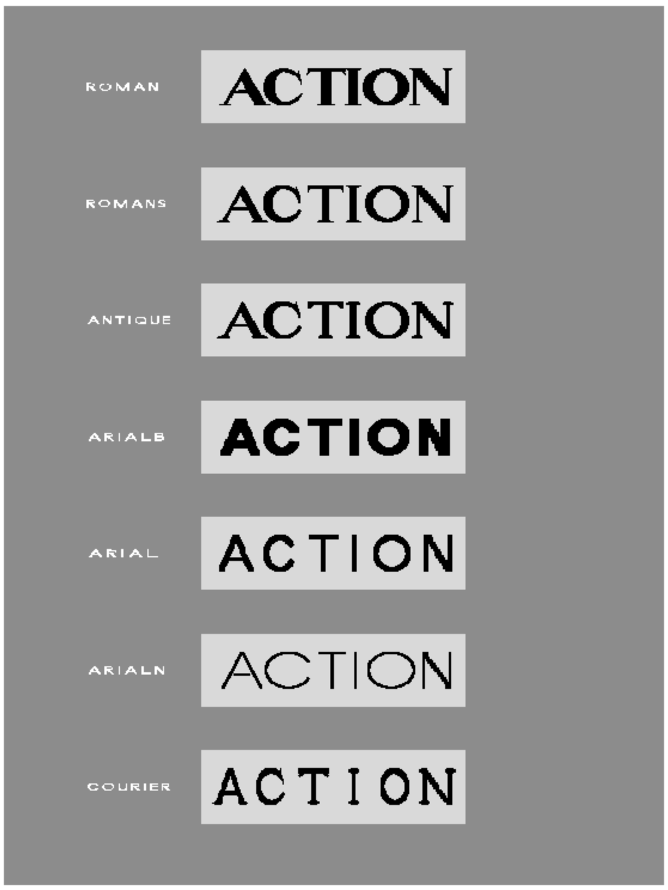

Figure B.1: C5 fonts

```
#define Roman_Draft_S  {-1.0,1.0,0.15,0.4 ,0.15,0.25,0.0,0.6 }
#define Roman_Letter_S {-2.0,1.0,0.15,0.4 ,0.15,0.25,0.0,0.6 }
#define Roman_Ultra_S  {-3.0,1.0,0.15,0.4 ,0.15,0.25,0.0,0.6 }

#define Roman_Draft_D  {-1.0,1.0,0.2,0.4,0.2,0.3,0.0,0.4 }
#define Roman_Letter_D {-2.0,1.0,0.2,0.4,0.2,0.3,0.0,0.4 }
#define Roman_Ultra_D  {-3.0,1.0,0.2,0.4,0.2,0.3,0.0,0.4 }

#define Roman_Draft_UD  {-1.0,1.0,0.25,0.4,0.25,0.35,0.0,0.1 }
#define Roman_Letter_UD {-2.0,1.0,0.25,0.4,0.25,0.35,0.0,0.1 }
#define Roman_Ultra_UD  {-3.0,1.0,0.25,0.4,0.25,0.35,0.0,0.1 }

#define Romans_Draft_UC  {-1.0,1.0,0.1 ,0.25,0.1 ,0.15,0.0,0.8}
#define Romans_Letter_UC {-2.0,1.0,0.1 ,0.25,0.1 ,0.15,0.0,0.8}
#define Romans_Ultra_UC  {-3.0,1.0,0.1 ,0.25,0.1 ,0.15,0.0,0.8}

#define Romans_Draft_C  {-1.0,1.0,0.1 ,0.3,0.1 ,0.2,0.0,0.7 }
#define Romans_Letter_C {-2.0,1.0,0.1 ,0.3,0.1 ,0.2,0.0,0.7 }
#define Romans_Ultra_C  {-3.0,1.0,0.1 ,0.3,0.1 ,0.2,0.0,0.7 }

#define Romans          {-1.0,1.0,0.1 ,0.3,0.15,0.25,0.0,0.6 }
#define Romans_Draft_S  {-1.0,1.0,0.1 ,0.3,0.15,0.25,0.0,0.6 }
#define Romans_Letter_S {-2.0,1.0,0.1 ,0.3,0.15,0.25,0.0,0.6 }
#define Romans_Ultra_S  {-3.0,1.0,0.1 ,0.3,0.15,0.25,0.0,0.6 }

#define Romans_Draft_D  {-1.0,1.0,0.2,0.3,0.2,0.3,0.0,0.4 }
#define Romans_Letter_D {-2.0,1.0,0.2,0.3,0.2,0.3,0.0,0.4 }
#define Romans_Ultra_D  {-3.0,1.0,0.2,0.3,0.2,0.3,0.0,0.4 }

#define Romans_Draft_UD  {-1.0,1.0,0.2,0.3,0.2,0.3,0.0,0.2 }
#define Romans_Letter_UD {-2.0,1.0,0.2,0.3,0.2,0.3,0.0,0.2 }
#define Romans_Ultra_UD  {-3.0,1.0,0.2,0.3,0.2,0.3,0.0,0.2 }

#define Antique_Draft_C  {-1.0,1.0,0.13,0.3,0.13,-0.28,0.0,0.76}
#define Antique_Letter_C {-2.0,1.0,0.13,0.3,0.13,-0.28,0.0,0.76}
#define Antique_Ultra_C  {-3.0,1.0,0.13,0.3,0.13,-0.28,0.0,0.76}

#define Antique          {-1.0,1.0,0.13,0.3,0.13,-0.31 ,0.0,0.6}
#define Antique_Draft_S  {-1.0,1.0,0.13,0.3,0.13,-0.31 ,0.0,0.6}
#define Antique_Letter_S {-2.0,1.0,0.13,0.3,0.13,-0.31 ,0.0,0.6}
#define Antique_Ultra_S  {-3.0,1.0,0.13,0.3,0.13,-0.31 ,0.0,0.6}
```

```
#define Antique_Draft_D  {-1.0,1.0,0.13,0.3 ,0.13,-0.32,0.0,0.4}
#define Antique_Letter_D {-2.0,1.0,0.13,0.3 ,0.13,-0.32,0.0,0.4}
#define Antique_Ultra_D  {-3.0,1.0,0.13,0.3 ,0.13,-0.32,0.0,0.4}

#define Antique_Draft_UD  {-1.0,1.0,0.13,0.3,0.13,-0.33,0.0,0.1}
#define Antique_Letter_UD {-2.0,1.0,0.13,0.3,0.13,-0.33,0.0,0.1}
#define Antique_Ultra_UD  {-3.0,1.0,0.13,0.3,0.13,-0.33,0.0,0.1}

/* Arial gorda bold */
#define Arialb_Draft_UC  {-1.0,0.0,0.4,0.4,0.4,0.0,0.0,0.85}
#define Arialb_Letter_UC {-2.0,0.0,0.4,0.4,0.4,0.0,0.0,0.85}
#define Arialb_Ultra_UC  {-3.0,0.0,0.4,0.4,0.4,0.0,0.0,0.85}

#define Arialb_Draft_C  {-1.0,0.0,0.4,0.4,0.4,0.0,0.0,0.70}
#define Arialb_Letter_C {-2.0,0.0,0.4,0.4,0.4,0.0,0.0,0.70}
#define Arialb_Ultra_C  {-3.0,0.0,0.4,0.4,0.4,0.0,0.0,0.70}

#define Arialb          {-1.0,1.0,0.4,0.4,0.4,0.0,0.0,0.50}
#define Arialb_Draft_S  {-1.0,1.0,0.4,0.4,0.4,0.0,0.0,0.50}
#define Arialb_Letter_S {-2.0,1.0,0.4,0.4,0.4,0.0,0.0,0.50}
#define Arialb_Ultra_S  {-3.0,1.0,0.4,0.4,0.4,0.0,0.0,0.50}

#define Arialb_Draft_D  {-1.0,1.0,0.42,0.42,0.42,0.00,0.0,0.3 }
#define Arialb_Letter_D {-2.0,1.0,0.42,0.42,0.42,0.00,0.0,0.3 }
#define Arialb_Ultra_D  {-3.0,1.0,0.42,0.42,0.42,0.00,0.0,0.3 }

#define Arialb_Draft_UD  {-1.0,1.0,0.42,0.42,0.42,0.00,0.0,0.0 }
#define Arialb_Letter_UD {-2.0,1.0,0.42,0.42,0.42,0.00,0.0,0.0 }
#define Arialb_Ultra_UD  {-3.0,1.0,0.42,0.42,0.42,0.00,0.0,0.0 }

/* Arial standard  */
#define Arial_Draft_UC  {-1.0,0.0,0.2,0.2,0.2,0.0,0.0,0.90}
#define Arial_Letter_UC {-2.0,0.0,0.2,0.2,0.2,0.0,0.0,0.90}
#define Arial_Ultra_UC  {-3.0,0.0,0.22,0.22,0.20,0.0,0.0,0.90}

#define Arial_Draft_C  {-1.0,0.0,0.2,0.2,0.2,0.0,0.0,0.7 }
#define Arial_Letter_C {-2.0,0.0,0.2,0.2,0.2,0.0,0.0,0.7 }
#define Arial_Ultra_C  {-3.0,0.0,0.22,0.22,0.20,0.0,0.0,0.7 }

#define Arial_Draft_S  {-1.0,1.0,0.2,0.2,0.2,0.0,0.0,0.60}
#define Arial_Letter_S {-2.0,1.0,0.2,0.2,0.2,0.0,0.0,0.60}
#define Arial_Ultra_S  {-3.0,1.0,0.2,0.2,0.2,0.0,0.0,0.60}
```

```
/* This is the default font  */
#define Arial           {-1.0,1.0,0.22,0.22,0.22,0.00,0.0,0.3 }
#define Arial_Draft_D   {-1.0,1.0,0.22,0.22,0.22,0.00,0.0,0.3 }
#define Arial_Letter_D  {-2.0,1.0,0.22,0.22,0.22,0.00,0.0,0.3 }
#define Arial_Ultra_D   {-3.0,1.0,0.21,0.21,0.21,0.00,0.0,0.3 }

#define Arial_Draft_UD   {-1.0,0.0,0.30,0.30,0.30,0.00,0.0,0.1}
#define Arial_Letter_UD  {-2.0,0.0,0.30,0.30,0.30,0.00,0.0,0.1}
#define Arial_Ultra_UD   {-3.0,0.0,0.30,0.30,0.30,0.00,0.0,0.1}

/* Arial delgada  */
#define Arialn_Draft_C   {-1.0,0.0,0.09,0.09,0.08,0.0,0.0,0.75}
#define Arialn_Letter_C  {-2.0,0.0,0.09,0.09,0.08,0.0,0.0,0.75}
#define Arialn_Ultra_C   {-3.0,0.0,0.07,0.07,0.06,0.0,0.0,0.75}

#define Arialn           {-1.0,0.0,0.09,0.09,0.08,0.0,0.0,0.55}
#define Arialn_Draft_S   {-1.0,0.0,0.09,0.09,0.08,0.0,0.0,0.55}
#define Arialn_Letter_S  {-2.0,0.0,0.09,0.09,0.08,0.0,0.0,0.55}
#define Arialn_Ultra_S   {-3.0,0.0,0.07,0.07,0.06,0.0,0.0,0.55}

#define Arialn_Draft_D   {-1.0,0.0,0.09,0.09,0.08,0.0,0.0,0.25}
#define Arialn_Letter_D  {-2.0,0.0,0.09,0.09,0.08,0.0,0.0,0.25}
#define Arialn_Ultra_D   {-3.0,0.0,0.07,0.07,0.06,0.0,0.0,0.25}

/* Courier  */
#define Courier_Draft_C   {-1.0,3.0,0.2 ,0.2 ,0.2 ,-0.4,0.0,0.5}
#define Courier_Letter_C  {-2.0,3.0,0.2 ,0.2 ,0.2 ,-0.4,0.0,0.5}
#define Courier_Ultra_C   {-3.0,3.0,0.22,0.22,0.20,-0.4,0.0,0.5}

#define Courier_Draft_S   {-1.0,3.0,0.2,0.2,0.2,-0.47,0.0,0.4 }
#define Courier_Letter_S  {-2.0,3.0,0.2,0.2,0.2,-0.47,0.0,0.4 }
#define Courier_Ultra_S   {-3.0,3.0,0.2,0.2,0.2,-0.47,0.0,0.4 }

#define Courier           {-1.0,3.0,0.24,0.24,0.24,-0.5,0.0,0.2}
#define Courier_Draft_D   {-1.0,3.0,0.24,0.24,0.24,-0.5,0.0,0.2}
#define Courier_Letter_D  {-2.0,3.0,0.24,0.24,0.24,-0.5,0.0,0.2}
#define Courier_Ultra_D   {-3.0,3.0,0.24,0.24,0.24,-0.5,0.0,0.2}

#define Courier_Draft_UD   {-1.0,3.0,0.30,0.30,0.30,-0.5,0.0,0.1}
#define Courier_Letter_UD  {-2.0,3.0,0.30,0.30,0.30,-0.5,0.0,0.1}
#define Courier_Ultra_UD   {-3.0,3.0,0.30,0.30,0.30,-0.5,0.0,0.1}
```

Appendix C

The C5 Graphics Library

C.1 The Graphics Library

1. `opm_page`

   ```
   Port_List opm_page(Color, Color, Color, ..., NULL)
   ```

   `opm_page` returns a port list with a `Right` oriented port. The size of the port is the entire page and the color list is constructed with the `Color` arguments that are not equal to `NULL`. The function accepts up to 12 arguments. (see `opm_Page` for constructing a page with an unbounded color list).

2. `opm_Page`

   ```
   Port_List opm_Page(Color_List)
   ```

   `opm_Page` returns a port list with a `Right` oriented port. The size of the port is the entire page and the color list is the argument. The constructor of color lists is `LCC` so `LCC(Red, LCC(Green,NULL))` is a valid color list of two elements.

3. `opm_print`

   ```
   void opm_print(Port_List)
   ```

   `opm_print` translates the port list argument into a PostScript format file using the standard output.

4. `opm_scale`

```
Port_List opm_scale(double left, double right,
             double up, double down,Port_List)
0.0 <= left  <=1.0
0.0 <= right <=1.0
0.0 <= up    <=1.0
0.0 <= down  <=1.0
```

`opm_scale` scales the ports of the argument according to the values of the arguments `left`, `right`, `up` and `down`.

5. *opm_rot*

```
Port_List opm_rot(int rotnr,Port_List)
```

`opm_rot` rotates `rotnr` $\times \Pi/2$ the ports of the list argument.

6. `opm_image_cons`

```
Port_List opm_image_cons(DTP,Port_List)
```

`opm_image_cons` returns an image (a port list) which is the graphic representation of the object member of the DPT argument printed on a page (the port list argument).

7. `opm_set_color`

```
Port_List opm_set_color(int color_idx,Port_List)
```

`opm_set_color` sets the current color of the ports of the list argument according to the `color_idx` value. if `color_idx` is greater than the color list length of the port then the current color is `White`.

8. `opm_colors`

```
DT_typedef struct{double x2,y1,x1,y2;} dp2;
DT_typedef struct OPMTON {
          dp2 scale;
          struct{ /* discr union  */
                  union{
                  int{0,10} bg;
                  struct OPMTON *next;
                  } un;
          int{1,0}  idx;
```

```
                } du;
        }{0} *Color_Serie;

      Color_Serie opm_colors(int mode,int tones_nr,
                             double coef1,double coef2)
```

`opm_colors` constructs a list of color scaled rectangles. The `mode` argument sets the scaling mode:

- `mode=0` sets the four sides of the rectangle for decreasing concentric scaling.
- `mode=1` sets the right side of the rectangle for decreasing concentric scaling.
- `mode=2` sets the right and down sides of the rectangle for decreasing concentric scaling.
- `mode=3` sets the right, down and up sides of the rectangle for decreasing concentric scaling.
- `mode=4` sets the left and right sides of the rectangle for left to right sequencing scaling.
- `mode=5` sets the up and down sides of the rectangle for up to down sequencing scaling.
- `mode=10` sets the right and left sides of the rectangle for decreasing concentric scaling.
- `mode=20` sets the up and down sides of the rectangle for decreasing concentric scaling.

The `tones_nr` argument defines the length of the color list and the `coef1` and `coef2` arguments are the scaling factor of the active and inactive sides respectively. The standard values of these arguments are 1.0 and 0.0.

9. `opm_col2col`

```
      Color opm_col2col(Color from, Color to, int tones_nr)
```

`opm_col2col` is a compressed notation for the colors of a port. The function represents a color serie starting from `from` to `to` of `tones_nr` tones.

10. `rgb`

```
Color  rgb(int, int, int)
```

The function rgb constructs a color object according to the values of the red, green and blue arguments. The range of the arguments is (0,255).

The White color is represented by rgb(0,0,0) and Black by rgb(255,255,255).

11. opm_bcurv4

```
Port_List opm_bcurv4(int rec_nr,
            double y1_U,double y2_U,double y3_U,double y4_U,
            double y1_D,double y2_D,double y3_D,double y4_D,
            Port_List pl)
```

The function opm_bcurv4 paints the shape limited by two Bezier curves of four points where the upper curve is defined by (0.0,y1_U), (0.33,y2_U), (0.66,y3_U), (1.0,y4_U) and the lower curve by (0.0,y1_D), (0.33,y2_D), (0.66,y3_D), (1.0,y4_D). The visible range of the y_... arguments is (0.0,1.0). rec_nr is the resolution level where 10 is the top manual resolution and -1, -2 , -3 and -4 are the automatic resolution for the draft, standard, high and ultra level respectively.

Figure C.1 shows the image produced by the four opm_bcurv4 examples of the next program:

```
main(){
    Port_List lp=opm_page(Black,Gray65,NULL),lp1,lp2,lp3,lp4;
    lp1=opm_scale(1.0,0.45,0.9,0.45,lp);
    lp2=opm_scale(0.45,1.0,0.9,0.45,lp);
    lp3=opm_scale(1.0,0.45,0.45,0.9,lp);
    lp4=opm_scale(0.45,1.0,0.45,0.9,lp);
    opm_print(opm_set_color(1,opm_concat(lp1,opm_concat(lp2,
       opm_concat(lp3,lp4)))));
    opm_print(opm_bcurv4( 4, 1.0, 0.2, 1.0, 0.0,
                            1.0, 0.0, 0.8, 0.0, lp1));
    opm_print(opm_bcurv4(-4, 1.0, 0.2, 1.0, 0.0,
                            1.0, 0.0, 0.8, 0.0, lp2));
    opm_print(opm_bcurv4(-3, 0.4, 1.2, 1.2, 0.4,
                            0.2, 1.0, 1.0, 0.2, lp3));
    opm_print(opm_bcurv4(-4, 1.0, 0.35,0.35,1.0,
                            0.5, 0.5, 0.5, 0.5, lp4));
    }
```

There is also a 5 points version called opm_bcurv5.

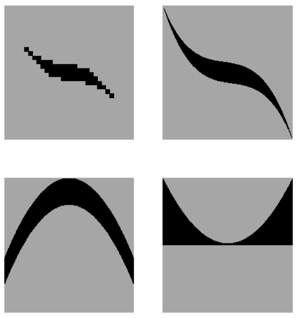

Figure C.1: 4 opm_bcurv4 examples

12. `opm_bcurv4p`

```
Port_List opm_bcurv4p(int rec_nr,
            double r1_U,double r2_U,double r3_U,double r4_U,
            double r1_D,double r2_D,double r3_D,double r4_D,
            Port_List pl)
```

The function `opm_bcurv4p` paints the surface limited by two Bezier curves of four points where the upper curve is defined in polar coordinates by (Π/2,r1_U), (0.66 × Π/2,r2_U), (0.33 × Π/2,r3_U), (0.0,r4_U) and the lower curve by (Π/2,r1_D), (0.66×Π/2,r2_D), (0.33×Π/2,r3_D),(0.0,r4_D). The visible range of the radius `r_...` arguments is (0.0,1.0). `rec_nr` is the resolution level where 10 is the top manual resolution and -1, -2 , -3 and -4 are the automatic resolution for the draft, standard, high and ultra level respectively.

Figure C.2 shows the image produced by the four `opm_bcurv4p` examples of the next program:

```
main(){
    Port_List lp=opm_page(Black,Gray65,NULL),lp1,lp2,lp3,lp4;
    lp1=opm_scale(1.0,0.45,0.9,0.45,lp);
    lp2=opm_scale(0.45,1.0,0.9,0.45,lp);
    lp3=opm_scale(1.0,0.45,0.45,0.9,lp);
    lp4=opm_scale(0.45,1.0,0.45,0.9,lp);
opm_print(opm_set_color(1,opm_concat(lp1,opm_concat(lp2,
   opm_concat(lp3,lp4)))));
opm_print(opm_bcurv4p( 4, 1.0, 0.2, 1.0, 0.0,
                        1.0, 0.0, 0.8, 0.0, lp1));
opm_print(opm_bcurv4p(-4, 1.0, 0.2, 1.0, 0.0,
                        1.0, 0.0, 0.8, 0.0, lp2));
opm_print(opm_bcurv4p(-3, 0.4, 1.2, 1.2, 0.4,
                        0.2, 1.0, 1.0, 0.2, lp3));
opm_print(opm_bcurv4p(-4, 1.0, 0.35,0.35,1.0,
                        0.5, 0.5, 0.5, 0.5, lp4));
}
```

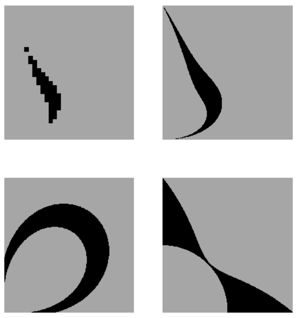

Figure C.2: 4 opm_bcurv4p examples

There is also a 5 points version called `opm_bcurv5p`.

13. `opm_ellipsis`

```
Port_List opm_ellipsis(int rec_nr,double ring,double elipse,
                     double incl1,double incl2,Port_List pl)
    -3 <= rec_nr <= 12
   0.0 <= ring   <= 1.0
   0.0 <= elipse <= 1.0
  -1.0 <= incl1  <= 1.0
  -1.0 <= incl2  <= 1.0
```

`opm_ellipsis` paints the surface delimited by the maximal ellipsis of the port and the ellipsis defined by `ring`. The `elipse` argument is an elliptical factor. `incl1` and `incl2` sets the inclination of the major and minor ellipsis respectively. `rec_nr` is the resolution level where 10 is the top manual resolution and -1, -2 , -3 and -4 are the automatic resolution for the draft, standard, high and ultra level respectively.

Figure C.3 shows the image produced by the four `opm_ellipsis` examples of the next program:

```
main(){
  Port_List lp=opm_page(Black,Gray65,NULL),lp1,lp2,lp3,lp4;
  lp1=opm_scale(1.0,0.45,0.9,0.45,lp);
  lp2=opm_scale(0.45,1.0,0.9,0.45,lp);
  lp3=opm_scale(1.0,0.45,0.45,0.9,lp);
  lp4=opm_scale(0.45,1.0,0.45,0.9,lp);
  opm_print(opm_set_color(1,opm_concat(lp1,opm_concat(lp2,
      opm_concat(lp3,lp4)))));
  opm_print(opm_ellipsis(-3, 0.0, 0.0, 0.0, 0.0,  lp1));
  opm_print(opm_ellipsis( 5, 0.90,0.40,0.90,0.20, lp2));
  opm_print(opm_ellipsis(-2, 0.55,0.20,0.30,-0.7, lp3));
  opm_print(opm_ellipsis(-4, 0.60,0.20,-0.8,1.00, lp4));
  }
```

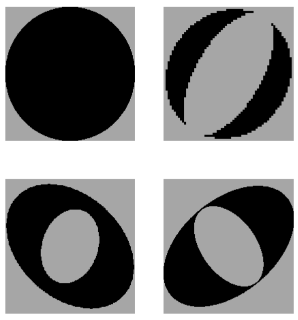

Figure C.3: 4 opm_ellipsis examples

14. `opm_sector`

```
Port_List opm_sector(int recnr,
```

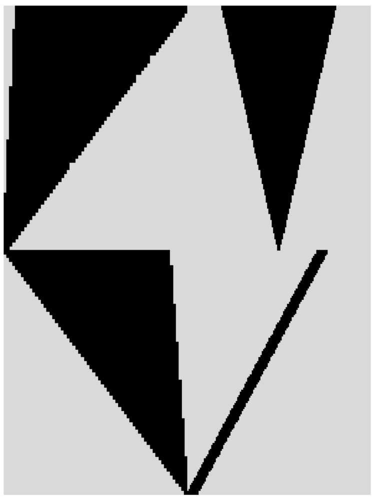

Figure C.4: 4 opm_plane_sector examples

```
                 double angle1, double x1,
                 double angle2, double x2,
                 Port_List lp);
   0.0 <= x1 <= 1.0
   0.0 <= x2 <= 1.0
```

`opm_sector` paints the intersection of the right plane sector defined by the line `line1` and the left plane sector defined by the line `line2`. `line1` and `line2` are defined by the X coordenates `x1` and `x2` with the angles in radians `angle1` and `angle2` respectively. `rec_nr` is the resolution level where 10 is the top manual resolution and -1, -2 , -3 and -4 are the automatic resolution for the draft, standard, high and ultra level respectively.

Figure C.4 shows the image produced by the four `opm_sector` examples of the next program:

```
main(){
Port_List lp;
lp=opm_page(Black,Gray85,Beige, NULL);
    opm_print(opm_set_color(1,lp));
opm_print(opm_sector(6, M_PI_2-0.07,0.00,
                        M_PI_4     ,0.00,
                        opm_scale(1.0,0.5,1.0,0.5,lp)));
opm_print(opm_sector(-1,M_PI_2+0.3, 0.50,
                        M_PI_2-0.3, 0.50,
                            opm_scale(0.5,1.0,1.0,0.5,lp)));
    opm_print(opm_sector(6, M_PI_2+M_PI_4, 1.00,
                            M_PI_2+0.1   , 1.00,
                            opm_scale(1.0,0.5,0.5,1.0,lp)));
    opm_print(opm_sector(-1,0.95, 0.00,
                            0.95, 0.05,
                            opm_scale(0.5,1.0,0.5,1.0,lp)));
    }
```

C.2 The specification of C5_image_cons.

Array type is denoted $T\ [n]$ where n is the size and T the type of the elements of the array.

rT_i denotes a recursive pointer type.

Pointer type is denoted $*\ T$ where T is the type of the referenced object.

Recursive pointer type is denoted $r*\ T$ where T is the type of the referenced object.

$$image\_cons:\ DPT \times Port\_List\ \rightarrow\ Port\_List$$

$$\begin{aligned}
image\_cons(<INT, i>, lp) &= selsplit(2, i+1, lp)\\
image\_cons(<INT\ TIE\{m,n\}, i>, lp) &= selsplit(n+1-m, i+1-m, lp)\\
image\_cons(<DOUBLE, d>, lp) &= partition(d, lp)\\
image\_cons(<DOUBLE&\\
TIE\{r,s: r>s\}, d>, lp) &= lp\\
image\_cons(<DOUBLE&\\
TIE\{r,s\}, d>, lp) &= partition((d-r)/(s-r), lp)\\
image\_cons(<CHAR, c>, lp) &= font_rep(Roman_Font, c, lp)\\
image\_cons(<CHAR\ TIE, c>, lp) &= font_rep(TIE, c, lp)\\
image\_cons(<STRUCT\{&\\
UNION\{T_1,..,T_n\}, INT\},&\\
a_i>, lp) &= image\_cons(<T_i, a_i>,\\
&= set\_color(i-1, lp))\\
image\_cons(<STRUCT\{&\\
UNION\{T_1,..,T_n\}, INT\}\ TIE\{c\},&\\
a_i>, lp) &= image\_cons(<T_i, a_i>,\\
&= set\_color(c*(i-1), lp))\\
image\_cons(<STRUCT\{T_1,..,T_n\}&
\end{aligned}$$

$$\begin{array}{rcl}
TIE\{r\},\{a_1,..,a_n\}>,lp) & = & inters(image\_cons(\\
 & & \quad < T_1, a_1 >, rot(0,lp)), \\
 & & inters(image\_cons(\\
 & & \quad < T_2, a_2 >, rot(r,lp)), \\
 & & ... \\
 & & image\_cons(< T_n, a_n >, \\
 & & \quad rot(r*(n-1),lp) \; ... \;) \\
\multicolumn{3}{l}{image\_cons(< STRUCT\{T_1,..,r* \; T_i,..,T_n\}} \\
TIE\{r\},\{a_1,..,a_n\}>,lp) & = & concat(image\_cons(\\
 & & \quad < T_1, a_1 >, rot(0,lp)), \\
 & & concat(image\_cons(\\
 & & \quad < T_2, a_2 >, rot(r,lp)), \\
 & & ... \\
 & & image\_cons(< T_n, a_n >, \\
 & & \quad rot(r*(n-1),lp) \; ... \;) \\
image\_cons(< CHAR \; [n] \; , str >, lp) & = & str\_rep(< CHAR \; [n] \; , str >, lp) \\
image\_cons(< T \; [n] \; TIE\{r,s,t:s>t\}, & & \\
\{a_0,..,a_{n-1}\}>,lp) & = & concat(image\_cons(\\
 & & \quad < T_0, a_0 >, rot(r,lp)), \\
 & & ... \\
 & & image\_cons(< T_{n-1}, a_{n-1} >, \\
 & & \quad rot(r,lp) \; ... \;) \\
image\_cons(< T \; [n] \; TIE\{r,s,t\}, & & \\
\{a_0,..,a_{n-1}\}>,lp) & = & concat(image\_cons(\\
 & & \quad < T_s, a_s >, rot(r, elem(t,lp))), \\
 & & ... \\
 & & image\_cons(< T_t, a_t >, \\
 & & \quad rot(r, elem(t,lp)) \; ... \;) \\
image\_cons(< CHAR \; * \; , str >, lp) & = & str\_rep(< CHAR \; * \; , str >, lp) \\
image\_cons(< T \; *, p_a >, lp) & = & image\_cons(< T, a >, rot(0,lp)) \\
image\_cons(< T \; * \; TIE\{r\}, p_a >, lp) & = & image\_cons(< T, a >, rot(r,lp)) \\
image\_cons(< TYPEDEF \; T \; id, a >, lp) & = & image\_cons(< T, a >, lp) \\
image\_cons(< TYPEDEF \; T \; id & & \\
TIE\{r,c\}, a >, lp) & = & image\_cons(< T, a >, \\
 & & rot(r, set\_color * c, lp))) \\
\end{array}$$

$$image\_cons(<\_,\_>, lp) \quad = \quad lp$$

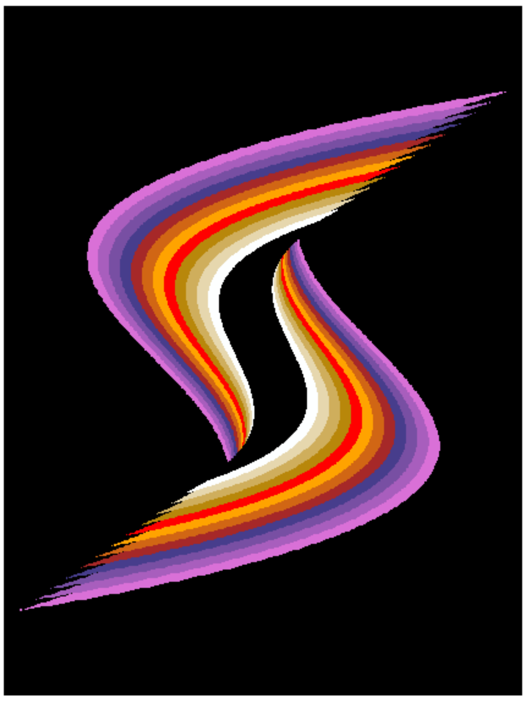

Figure C.5: The flying S.

C.3 An example using Bezier curves

The next program combines the functions `opm_colors`, `opm_co2col` and `opm_bcurv5` to generate a fantasy letter S :

```
main(){
    Color_Serie cs=opm_colors(20,SERIE_NR,1.0,0.33);
Port_List lp=opm_page(
        opm_col2col(Orchid, SlateBlue4,TONES_NR),
        opm_col2col(Brown ,Orange,   TONES_NR-1), Red,
        opm_col2col(DarkGoldenrod , White,   TONES_NR),
        opm_col2col(Black  , Black, TONES_NR),
        LightYellow1, Black , NULL);
opm_print(opm_page( Black ,NULL));
opm_print(opm_bcurv5(-1,1.0, 0.1,0.65,1.6,0.0,
                        1.0,-0.6,0.35,0.9,0.0,
          opm_image_cons(DT_pair(Color_Serie,cs),
          opm_rot(1,
          opm_scale(0.99,0.99,0.88,0.88,lp)))));
}
```

Figure C.5 shows the resulting image.

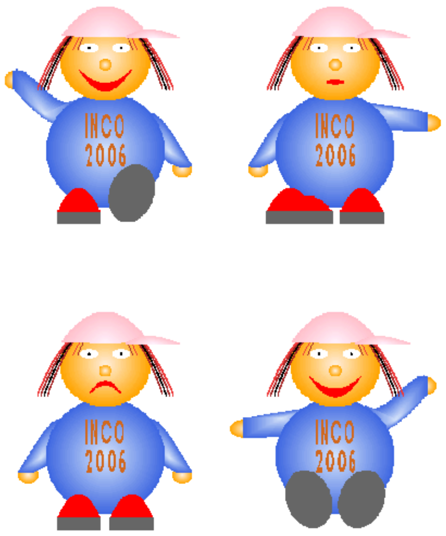

Figure C.6: The girl mascot.

C.4 The mascot of InCo 2006

In 2006, undergraduate students at InCo created 48 mascots programmed in C5. We present three of them to give an overview of this work.

C.4.1 The girl mascot.

Laura Cuadrado and Cecilia Stevenazzi produced a C5 program (194 lines) creating the mascot presented in Figure C.6.

C.4.2 The boy mascot.

Andrés Américo, Carlos Baptista and Fernando Martínez produced a C5 program (282 lines) creating the mascot presented in Figure C.7.

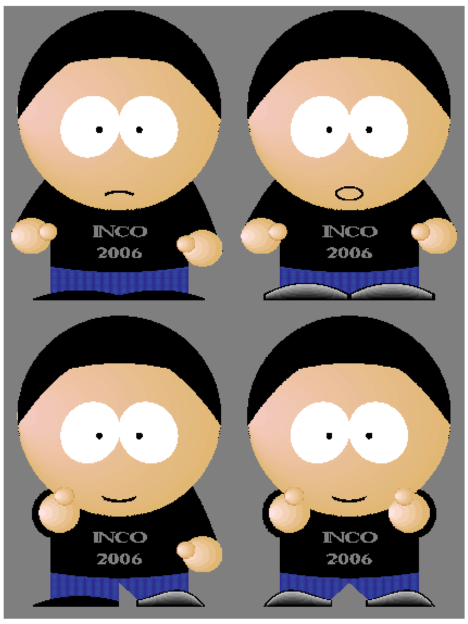

Figure C.7: The South Park mascot.

C.4.3 The green monster.

iWalmar Laiolo , Nicolás Gerolami and Marcelo Perelmutter produced a C5 program (334 lines) creating the mascot presented in Figure C.8.

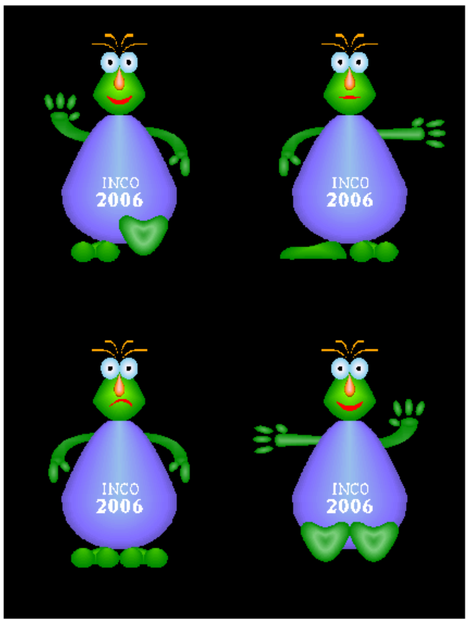

Figure C.8: The green monster mascot.

Appendix D

Specification of the C5 structural equality.

D.1 Type equality

Array type is denoted $a\ [n]$ where n is the size and a the type of the elements of the array.

Pointer type is denoted $*\ a$ where a is the type of the referenced object.

Recursive pointer type is denoted $r*\ a$ where a is the type of the referenced object.

Let $m, n : int$; $a, b, t, a_i, b_i : type$; $id : identifier$; $< type, object >: DPT$;

$$typeSeq : \ DPT \ \times \ DPT \ \rightarrow \ int$$

$$\begin{aligned}
typeSeq(< TYPEDEF\ t\ id, \_ >, < \_, \_ >) &= typeSeq(< t, \_ >, < \_, \_ >)\\
typeSeq(< \_, \_ >, < TYPEDEF\ t\ id, \_ >) &= typeSeq(< \_, \_ >, < t, \_ >)\\
typeSeq(< INT, \_ >, < INT, \_ >) &= 1\\
typeSeq(< CHAR, \_ >, < CHAR, \_ >) &= 1\\
typeSeq(< DOUBLE, \_ >, < DOUBLE, \_ >) &= 1\\
typeSeq(< STRUCT\{a_1, .., a_n\}, \_ >, &\\
< STRUCT\{b_1, .., b_m\}, \_ >) &= m == n\ \&\&\\
& \quad typeSeq(< a_1, \_ >, < b_1, \_ >)\ \&\&\\
& \quad ...\\
& \quad typeSeq(< a_{n,>}, < b_m, \_ >)\\
typeSeq(< UNION\{a_1, .., a_n\}_{,>}, &\\
< UNION\{b_1, .., b_m\}, \_ >) &= m == n\ \&\&\\
& \quad typeSeq(< a_1, \_ >, < b_1, \_ >)\ \&\&
\end{aligned}$$

$$\begin{aligned}
& & & ... \\
& & & typeSeq(<a_n,\_>,<b_m,\_>) \\
typeSeq(<a\ [n],\_>,<b\ [m],\_>) &=& & m == n\ \&\& \\
& & & typeSeq(<a,\_>,<b,\_>) \\
typeSeq(<*\,a,\_>,<*\,b,\_>) &=& & typeSeq(<a,\_>,<b,\_>) \\
typeSeq(<r*\ a,\_>,<r*\ b,\_>) &=& & check\_table(<r*\ a,\_>, \\
& & & <r*\ b,\_>) \\
typeSeq(<r*\ a,\_>,<*\,b,\_>) &=& & 0 \\
typeSeq(<*\,a,\_>,<r*\ b,\_>) &=& & 0 \\
typeSeq(<a\ FUNCTION(a_1,..,a_n),\_>, & & & \\
<b\ FUNCTION(b_1..,b_m),\_>) &=& & m == n\ \&\& \\
& & & typeSeq(<a,\_>,<b,\_>)\ \&\& \\
& & & typeSeq(<a_1,\_>,<b_1,\_>)\ \&\& \\
& & & ... \\
& & & typeSeq(<a_n,\_>,<b_m,\_>) \\
typeSeq(<\_,\_>,<\_,\_>) &=& & 0
\end{aligned}$$

In the next function, $r*\ a$ denotes a recursive pointer type that is not member of the pointer table. The function *insert* inserts a pointer type in the pointer table.

$tr*\ a$ denotes a recursive pointer type which is member of the pointer table. This pointer has been inserted in the pointer table using the function *insert*.

$$check\_table:\ DPT\ \times\ DPT\ \rightarrow\ int$$

$$\begin{aligned}
check\_table(<r*\ a,\_>,<r*\ b,\_>) &=& & (insert(r*\ a)); \\
& & & (insert(r*\ a)); \\
& & & typeSeq(<a,\_>,<b,\_>) \\
check\_table(<tr*\ a,\_>,<r*\ b,\_>) &=& & (insert(r*\ b)); \\
& & & typeSeq(<a,\_>,<b,\_>) \\
check\_table(<r*\ a,\_>,<tr*\ b,\_>) &=& & (insert(r*\ a)); \\
& & & typeSeq(<a,\_>,<b,\_>) \\
check\_table(<tr*\ a,\_>,<tr*\ b,\_>) &=& & 1
\end{aligned}$$

D.2 Object equality

$$Seq : DPT \times DPT \rightarrow int$$

$$\begin{aligned}
Seq(<A,a>,<B,b>) &= typeSeq(<A,a>,<B,b>)\ \&\& \\
&\ seq(<A,a>,<B,b>)
\end{aligned}$$

$$seq : DPT \times DPT \rightarrow int$$

$$\begin{aligned}
seq(<TYPEDEF\ t\ id, obj>,<\_,\_>) &= seq(<t, obj>,<\_,\_>) \\
seq(<\_,\_>,<TYPEDEF\ t\ id, obj>) &= typeseq(<\_,\_>,<t, obj>) \\
seq(<INT, o_a>,<INT, o_b>) &= o_a == o_b \\
seq(<DOUBLE, o_a>,<DOUBLE, o_b>) &= o_a == o_b \\
seq(<CHAR, o_a>,<CHAR, o_b>) &= o_a == o_b \\
seq(<STRUCT\{UNION\{T_1,..,T_n\}, INT\}, & \\
\{\{a_1,..,a_n\}, i\}>, & \\
STRUCT\{UNION\{T_1,..,T_n\}, INT\}, & \\
\{\{a_1,..,a_n\}, j\}>) &= i == j\ \&\& \\
&\ seq(<T_i, a_i>,<T_j, b_j>) \\
seq(<STRUCT\{T_1,..,T_n\}, \{a_1,..,a_n\}>, & \\
<STRUCT\{T_1,..,T_n\}, \{b_1,..,b_n\}>) &= seq(<T_1, a_1>,<T_1, b_1>)\ \&\& \\
&\ ... \\
&\ seq(<T_n, a_n>,<T_n, b_n>) \\
seq(<*T, NULL>,<*T,\_>) &= 0 \\
seq(<*T,\_,<*T, NULL>) &= 0 \\
seq(<*char, s_a>,<*char, s_b>) &= strcmp(s_a, s_b) \\
seq(<*T, p^a>,<*T, p^b>) &= seq(<T, a>,<T, b>) \\
seq(<CHAR\ [n], s_a> & \\
<CHAR\ [n], s_b>) &= strcmp(s_a, s_b) \\
seq(<T\ [n], \{a_0,..,a_{n-1}\}>, & \\
<T\ [n], \{b_0,..,b_{n-1}\}>) &= seq(<T, a_0>,<T, b_0>)\ \&\& \\
&\ ... \\
&\ seq(<T, a_{n-1}>,<T, b_{n-1}>) \\
seq(<UNION,\_>,<UNION,\_>) &= (Undefined)\ 0
\end{aligned}$$

$$\begin{aligned} seq(<FUNCTION, \_>, <FUNCTION, \_>) &= (Undefined)\ 0 \\ seq(<\_, \_>, <\_, \_>) &= 0 \end{aligned}$$

Appendix E

The syntax of C5.

C5 syntax is a C grammar [16] with few modifications. In particular, the syntax of TIEs is described by the *typeinf* rules.

High level definitions.

$$\begin{array}{rcl}
program & := & ext\_def\_list; \\
ext\_def\_list & := & ext\_def\_list\ ext\_def \\
 & | & /*\ epsilon\ */; \\
ext\_def & := & opt\_specifiers\ ext\_decl\_list\ SEMI \\
 & | & opt\_specifiers\ SEMI \\
 & | & opt\_specifiers\ func\_decl\ def\_list\ compound\_stmt; \\
ext\_decl\_list & := & ext\_decl \\
 & | & ext\_decl\_list\ COMMA\ ext\_decl; \\
ext\_decl & := & func\_decl \\
 & | & var\_decl \\
 & | & var\_decl\ EQUALL\ initializer; \\
opt\_specifiers & := & specifiers \\
 & | & /*\ empty\ */; \\
specifiers & := & classoconst\ ttype\ typeinf \\
 & | & ttype\ typeinf \\
 & | & class\ oconst\ type \\
 & | & type \\
 & | & class; \\
type & := & type\_specifier;
\end{array}$$

Specifiers

$$\begin{array}{rcl}
type\_specifier & := & atomict\_list \\
 & | & enum\_specifier \\
 & | & struct\_specifier; \\
atomict\_list & := & atomic\_specifier \\
 & | & atomict\_list\ atomic\_specifier; \\
atomic\_specifier & := & \\
atostypeinf; & & \\
class & := & CLASS; \\
ttype & := & TTYPE; \\
atos & := & TYPE; \\
oconst & := & \\
 & | & CONST;
\end{array}$$

Enumeration

$$\begin{array}{rcl}
enum\_specifier & := & ENUM\ name\ opt\_enum\_list \\
 & | & ENUM\ LC\ enumerator\_list\ RC; \\
opt\_enum\_list & := & LC\ enumerator\_list\ RC \\
 & | & /*\ empty\ */; \\
enumerator\_list & := & enumerator \\
 & | & enumerator\_list\ COMMA\ enumerator; \\
enumerator & := & name \\
 & | & name\ EQUALL\ const\_expr;
\end{array}$$

Variable declarators

$$\begin{array}{rcl}
var\_decl & := & new\_name \\
 & | & var\_decl\ LP\ RP \\
 & | & var\_decl\ LP\ var\_list\ RP \\
 & | & var\_decl\ LB\ RB\ typeinf \\
 & | & var\_decl\ LB\ const\_expr\ RB\ typeinf
\end{array}$$

$$\begin{array}{rcl}
 & | & STAR\ var\_decl \\
 & | & STAR\ typeinf\ var\_decl \\
 & | & LP\ var\_decl\ RP; \\
new\_name & := & NAME; \\
name & := & NAME;
\end{array}$$

Function declarators

$$\begin{array}{rcl}
func\_decl & := & STAR\ func\_decl \\
 & | & func\_decl\ LB\ RB \\
 & | & func\_decl\ LB\ const\_expr\ RB \\
 & | & LP\ func\_decl\ RP \\
 & | & func\_decl\ LP\ RP \\
 & | & new\_name\ LP\ RP \\
 & | & new\_name\ LP\ name\_list\ RP \\
 & | & new\_name\ LP\ var\_list\ RP;
\end{array}$$

$$\begin{array}{rcl}
name\_list & := & new\_name \\
 & | & name\_list\ COMMA\ new\_name; \\
var\_list & := & param\_declaration \\
 & | & var\_list\ COMMA\ param\_declaration; \\
param\_declaration & := & type\ var\_decl \\
 & | & ttype\ var\_decl \\
 & | & abstract\_decl \\
 & | & ELLIPSIS;
\end{array}$$

Abstract declarators

$$\begin{array}{rcl}
abstract\_decl & := & abs\_decl \\
 & | & type\ abs\_decl \\
 & | & CONST\ type\ abs\_declo name \\
 & | & ttype\ abs\_decl \\
 & | & CONST\ ttype\ abs\_decl\ oname; \\
oname & := &
\end{array}$$

$$\begin{array}{rcl}
 & \mid & name; \\
abs\_decl & := & /*\,epsilon\,*/ \\
 & \mid & CONST\ abs\_decl \\
 & \mid & LP\ abs\_decl\ RP\ LP\ RP \\
 & \mid & LP\ abs\_decl\ RP\ LP\ var\_list\ RP \\
 & \mid & STAR\ abs\_decl \\
 & \mid & abs\_decl\ LB\ RB\ typeinf \\
 & \mid & abs\_decl\ LB\ const\_expr\ RB \\
 & \mid & LP\ abs\_decl\ RP;
\end{array}$$

Structures

$$\begin{array}{rcl}
struct\_specifier & := & STRUCT\ opt\_tag\ LC\ def\_list\ RC\ typeinf \\
 & \mid & UNION\ opt\_tag\ LC\ def\_list\ RC\ typeinf \\
 & \mid & STRUCT\ tag \\
 & \mid & UNION\ tag; \\
opt\_tag & := & tag \\
 & \mid & /*\,empty\,*/; \\
tag & := & name \\
 & \mid & ttype; \\
typeinf & := & /*\,C5TIEdeclaration\,*/ \\
 & \mid & LC\ initializer\_list\ opt\_comma\ RC;
\end{array}$$

Local variables and function args.

$$\begin{array}{rcl}
def\_list & := & def\_listdef \\
 & \mid & /*\,epsilon\,*/; \\
def & := & specifiers\ decl\_list\ SEMI \\
 & \mid & specifiers\ SEMI; \\
decl\_list & := & decl \\
 & \mid & decl\_list\ COMMA\ decl; \\
decl & := & func\_decl \\
 & \mid & var\_decl \\
 & \mid & var\_decl\ EQUALL\ initializer
\end{array}$$

```
                     |   var_decl COLON const_expr %prec COMMA
                     |   COLON const_expr %prec COMMA;
      initializer   :=   expr %prec COMMA
                     |   LC initializer_list opt_comma RC;
        opt_comma   :=   /* empty */
                     |   COMMA;
 initializer_list   :=   initializer
                     |   initializer_list COMMA initializer;
```

Statements

```
compound_stmt  :=  LC local_defs stmt_list RC;
   local_defs  :=  def_list;
    stmt_list  :=  stmt_list statement
                |  /* epsilon */;
    statement  :=  expr SEMI
                |  compound_stmt
                |  RETURN SEMI
                |  RETURN expr SEMI
                |  BREAK SEMI
                |  CONTINUE SEMI
                |  SEMI
                |  GOTO target SEMI
                |  target COLON statement
                |  SWITCH LP expr RP compound_stmt
                |  CASE const_expr COLON
                |  DEFAULT COLON
                |  IFTHEN LP test RP statement
                |  IFTHEN LP test RP statement ELSE statement
                |  WHILE LP test RP statement
                |  FOR LP opt_expr SEMI test SEMI opt_expr RP statement
                |  DO statement WHILE LP test RP SEMI;
         test  :=  expr
                |  /* epsilon */;
       target  :=  name;
```

Expressions

```
      opt_expr  :=  expr
                 |  /* epsilon */;
    const_expr  :=  expr;
          expr  :=  expr COMMA non_comma_expr
                 |  non_comma_expr;
non_comma_expr  :=  non_comma_expr QUEST non_comma_expr
                    COLON non_comma_expr
                 |  non_comma_expr assignop non_comma_expr
                 |  non_comma_expr EQUALL non_comma_expr
                 |  or_expr;
       or_expr  :=  or_list;
       or_list  :=  or_list OROR and_expr
                 |  and_expr;
      and_expr  :=  and_list;
      and_list  :=  and_list ANDAND binary
                 |  binary;
        binary  :=  binary relop binary
                 |  binary equop binary
                 |  binary STAR binary
                 |  binary divop binary
                 |  binary shiftop binary
                 |  binary AND binary
                 |  binary XOR binary
                 |  binary OR binary
                 |  binary PLUS binary
                 |  binary MINUS binary
                 |  unary;
         unary  :=  LP expr RP
                 |  fcon
                 |  icon
                 |  name
                 |  string_const %prec COMMA
```

```
                |  SIZEOF LP string_const RP %prec SIZEOF
                |  SIZEOF LP expr RP %prec SIZEOF
                |  SIZEOF LP abstract_decl RP %prec SIZEOF
                |  LP abstract_decl RP unary %prec UNOP
                |  MINUS unary %prec UNOP
                |  unop unary
                |  unary incop
                |  incop unary
                |  AND unary %prec UNOP
                |  STAR unary %prec UNOP
                |  unary LB expr RB %prec UNOP
                |  unary structop name %prec STRUCTOP
                |  unary LP args RP
                |  unary LP RP;
     assignop  :=  ASSIGNOP;
        relop  :=  RELOP;
        equop  :=  EQUOP;
         unop  :=  UNOP;
        incop  :=  INCOP;
        divop  :=  DIVOP;
      shiftop  :=  SHIFTOP;
     structop  :=  STRUCTOP;
         icon  :=  ICON;
         fcon  :=  FCON;

 string_const  :=  string string_const
                |  string;
       string  :=  STRING;
         args  :=  non_comma_expr %prec COMMA
                |  non_comma_expr COMMA args;
```

.

Appendix F

Specification and code of C5_compil.

F.1 The specification of compil

Array type is denoted $T\ [n]$ where n is the size and T the type of the elements of the array.

Pointer type is denoted $*\ T$ where T is the type of the referenced object.

$TIE\{FUNCTION, A_1, .., A_n\}$ denotes a TIE where the first element is a function DPT.

$\{a_1, .., a_n\}$ denotes the fields of a `struct` or `union` object.

$\{a_0, .., a_n\}$ denotes the elements of an array object.

$T, T_0, .., T_n : type$; $id : identifier$; $< type, object >: DPT$;

$$compil :\ DPT\ \rightarrow\ DPT$$

$$\begin{aligned}
compil(< INT, \_ >) &= < INT, \_ > \\
compil(< CHAR, \_ >) &= < CHAR, \_ > \\
compil(< DOUBLE, \_ >) &= < DOUBLE, \_ > \\
compil(< "Token", \_ >) &= < "Token", \_ > \\
compil(< STRUCT\{ & \\
UNION\{T_1, .., T_n\}, INT\}, & \\
\{\{a_1, .., a_n\}, i\} >, &= compil(< T_i, a_i >) \\
compil(< STRUCT\{ & \\
UNION\{T_1, .., T_n\}, & \\
INT\}\ TIE, & \\
\{\{a_1, .., a_n\}, i\} >, &= actions(< T_i, a_i >)
\end{aligned}$$

$$\begin{aligned}
compil(<STRUCT\ TIE, \_>) &= actions(<STRUCT\ TIE, \_>)\\
compil(<STRUCT, \_>) &= <STRUCT, \_>\\
compil(<CHAR\ [n], \_>) &= <CHAR\ [n], \_>\\
compil(<T\ [n], \_>) &= <T\ [n], \_>\\
compil(<T\ [n] TIE, \_>) &= actions(<T\ [n], \_>)\\
compil(<T\ *\ TIE\{A_1, .., A_n\}, NULL>) &= A_1\\
compil(<T\ *\ , NULL>) &= <T\ *\ , NULL>\\
compil(<T\ *\ , p \rightarrow a>) &= compil(<T, a>)\\
compil(<TYPEDEF\ T\ id\ TIE, \_>) &= actions(\\
&\quad <TYPEDEF\ T\ id\ TIE, \_>)\\
compil(<TYPEDEF\ T\ id, \_>) &= compil(<T, \_>)\\
compil(<\_, \_>) &= <\_, \_>
\end{aligned}$$

$$actions:\ DPT \rightarrow DPT$$

$$\begin{aligned}
&actions(<\_\\
&TIE\{FUNCTION, A_1, .., A_n\},\\
&\_>) = fapply(FUNCTION,\\
&\qquad map\_tie(<\_\\
&\qquad\quad TIE\{FUNCTION, A_1, .., A_n\}, \_>,\\
&\qquad\quad \{A_1, .., A_n\})\\
&actions(<\_TIE, \_>) = hd(map\_tie(<\_TIE, \_>, TIE))
\end{aligned}$$

$$map\_tie:\ DPT \times DPT\_List \rightarrow DPT\_List$$

$$\begin{aligned}
map\_tie(<\_, \_>, nil) &= nil\\
map\_tie(dp, a : l) &= compil(getdp(dp, a)) :\\
&\quad map\_tie(dp, l)
\end{aligned}$$

$$getdp : \; DPT \;\times\; DPT \;\rightarrow\; DPT$$

$$\begin{aligned}
getdp(< A\,[n], \{a_0, .., a_{n-1}\} >, < INT, \{0 \le i < n\} >) &= < A, a_i > \\
getdp(< STRUCT\{UNION, INT\}, \_ >, & \\
< T, a >) &= < T, a > \\
getdp(< STRUCT\{A_1, .., A_n\}, \{a_1, .., a_n\} >, & \\
< CHAR *, name(A_i) >) &= < A_i, a_i > \\
getdp(< STRUCT\{A_1, .., A_n\}, \{a_1, .., a_n\} >, & \\
< CHAR *, str >) &= < CHAR *, str > \\
getdp(< \_, \_ >, < T, a >) &= < T, a >
\end{aligned}$$

F.2 The code of C5_compil.

```
DPT C5_compil(DPT dp){
    int i;
    if(!strncmp("Token",C5_gname(dp),5)) return(dp);
    switch(C5_gtype(dp)){
        case CHAR: case DOUBLE: case INT: return(dp);
        case STRUCT:
              if(isDUnion(dp)){   /* Discriminated Union */
                      if (C5_TIE_length(dp)==0)  /* No TIE */
                              return(C5_compil(C5_gos(C5_gos(dp,1),
                                      C5_gint(C5_gos(dp,2),0)+1  )));
                      else return(C5_scanfActions(dp));
                      }
              else{
                      if (C5_TIE_length(dp)==0) return(dp);
                      else return(C5_scanfActions(dp));
                      }
        case ARRAY:
              if (C5_TIE_length(dp)==0 ||
                      C5_gtype(C5_gos(dp,0))==CHAR) return(dp);
              else return(C5_scanfActions(fp,dp));
        case POINTER:
              if(C5_is_ptr_nul(dp) ||
                      C5_gtype(C5_gos(dp,1))==CHAR) return(dp);
        case TYPEDEF:
                if (C5_TIE_length(dp)>0) return(C5_scanfActions(dp));
                else return(C5_compil(C5_gos(dp,1)));
            case FUNCTION: return(dp);
            default: fprintf(fp,"Unknown type %s.\n",C5_gname(dp));
                     return(dp);
            }
        }

DPT  C5_scanfActions(DPT dp){     /*  */
       DPT tie1= dphd(C5_gtie(dp));  /* first element of the TIE */
       if(C5_isFunction(tie1))  /* function application */
             return(C5_fapply(tie1,
                 C5_map_tie(fp,dp, dptl(C5_gtie(dp)))));
       else return(dphd(C5_map_tie(dp,C5_gtie(dp))));
       }

DPT_list  C5_map_tie(DPT dp, DPT_list tie_ls){
       if(tie_ls==NULL) return(NULL);
       else{
             DPT auxdp= C5_compil(C5_gPtrdp(dp,dphd(tie_ls)));
             return(dpcons(auxdp,C5_map_tie(dp, dptl(tie_ls))));
             }
        }

DPT  C5_gPtrdp(DPT dp, DPT tie){
        DPT out;  int i;
        char *st;
        if(C5_gtype(dp)==ARRAY && C5_gtype(tie)==INT){
                int tie_idx=C5_gint(tie,-1);
                if(tie_idx<0 || tie_idx>=C5_gsize(dp)) return(dp);
                else return(C5_gos(dp,tie_idx));
                }
        if(C5_gtype(dp)==STRUCT && !isDUnion(dp) && C5_isString(tie)){
char *str= C5_gstr(tie, "C5_gPtrdp error");
                for(i=1;i<=C5_gcant(dp);i++)
                        if(!strcmp(str,C5_gname(C5_gos(dp,i))))
                                return(C5_gos(dp,i));
                return(tie);  /* no match */
                }
        return(tie); /* no string */
        }
```