AutoMap: A Data Structures Compiler for the Automatic Generation of MPI Data Structures Directly From C Code

Judith Ellen Devaney
NIST
Martial Michel
ESIAL
Jasper Peeters
University of Twente
Eric Baland
Institut National des Télécommuncations

April 27, 1997

Abstract

AutoMap is a compiler that automatically translates C structs into MPI datatypes. Its grammar recognizes flags in the code that tell which C structs require MPI datatypes. It then reads these structs and creates the MPI datatypes for them. There are few limitations on the C structs it can handle. For example, structs can contain pointers and arrays, as well as the usual data types. Additionally, the structs can be nested, creating nested datatypes.

The utility can be used standalone; or it can be used in conjunction with AutoLink. AutoLink is an MPI library for sending and receiving dynamic data structures.

Contents

List of Figures

Chapter 1

Automation Through a Designed Grammar

1.1 Introduction

Programming parallel computers is facilitated through the use of tools and libraries that automate mundane, tedious, and/or repetitious tasks. With the advent of the standard Message Passing Interface (MPI) [1], one now has available a powerful and flexible infrastructure on which to build these tools and libraries. And with the creation of tools, the programmer is freeer to think at a higher level about their program.

One abstraction used by programmers is data structures. Data structures are ubiquitous in programming. They enable grouping of information in a simple and intuitive way. MPI allows the sending and receiving of data structures in addition to the sending and receiving of the predefined data types. However, in order to use this capability, the user must create a new MPI data type. The general MPI datatype requires specification of: a sequence of basic datatypes, and a sequence of integer displacements [1]. The pair of sequences is the *type map* of the datatype, and the sequence of basic datatypes is the *type signature* of the datatype. This requires that the MPI library be provided with the exact layout in memory of the data structure. This is tedious and error prone. To get an example of what is required to create an MPI datatype, consider the constuction of one given in the Message Passing Interface (MPI) [1] standard document.

The structure to be converted is the following:

```
struct Partstruct
```

```
  {
    int class;    /* particle class */
    double d[6];  /* particle coordinates */
    char   b[7];  /* additional information */
  };
```

First the name of the MPI data structure is declared:

```
MPI_Datatype Particletype;
```

Then a typemap for the structure must be specified. This is accomplished as follows. The data types of each of the fields of the structure must be given, followed by the number of items of each field.

```
MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
int blocklen[3] = {1, 6, 7};
```

In order to get the addresses of the parts of the structure, as required by MPI, a base address and displacements for each of the types in the structure are needed. So variables are declared for these and then they are filled with calls to MPI routines as follows:

```
/* declare an array to contain the displacements */
MPI_Aint   displ[3];
/* base holds the base address of the structure       */
int        base;
/* compute displacements of structure components */
MPI_Address(particle, disp);
MPI_Address(particle[0].d, disp+1);
MPI_Address(particle[0].b, disp+2);
base = disp[0];
for(i = 0; i < 3; i++) disp[i] -= base;
```

The MPI type is then declared and committed.

```
MPI_Type_struct(3, blocklen, disp, type, &Particletype);
MPI_Type_commit(&Particletype);
```

There are also MPI issues that can come up that affect how the typemap is to be specified. See section 1.5 for a discussion of this. The detail required and tediousness of the operations make it a good candidate for automation. AutoMap [9] provides that automation.

AutoMap reads C code, takes the user-marked data structures, and outputs a file of MPI data types. The programmer then includes this file in their program in order to have access to the new MPI datatypes. The new MPI datatypes are named after their C structs making their use straightforward.

AutoMap can be used in two separate ways. It can be used as a stand alone utility for creating the MPI data types. Or it can be used in conjunction with AutoLink. AutoLink is an MPI library for sending and receiving dynamic data structures. The AutoLink library enables trees, linked lists, graphs or any other dynamic data structure to be sent to another process. The library creates the valid memory pointers on the receiving process.

1.2 An MPI Data Structures Compiler

A compiler implements a grammar. It is a useful abstraction that frees programmers from the details of machine code. It is also useful for operations that can be described by means of a grammar. Tools that can convert a grammar into a compiler exist. This further simplifies the process. AutoMap is the creation of a compiler from a grammar. The grammar is necessary to read the C structs and output the MPI datatypes corresponding to the C structs.

The compiler generator used here is yacc++ [4]. It is object oriented and comes with libraries that further simplify the process of building a compiler.

1.3 Basic concepts

The requirement is the construction of a *grammar*. To read this grammar and apply its rules, one has to build a compiler that reads an input (not necessarily a file), check if it matches the grammar rules and produce an output, as specified by the compiler writer. There are two parts to a compiler, a lexer and parser.

1.3.1 The lexer

The first part of the compiler is the *lexer* (lexical analyzer): It reads the input one character at a time and tries to recognize the tokens of the grammar. These tokens are minimal sequences of characters that have meaning as an entity. They can be reserved words in the grammar or allowable variable names, for example. The lexer reads the input and gives the tokens to the parser. If there is a wrong combination of characters—a combination that doesn't form a token—then it produces an error. Each time a token is completed and then passed to the parser, one says that the token is *reduced* by the lexer.

1.3.2 The parser

The input to the *parser* (syntax analyzer) is the stream of tokens that comes from the lexer. The first job of the parser is to group the tokens. The grouping is accomplished by the rules of the grammar. These rules are expressed by means of the BNF (Backus-Naur Form) notation, detailed in [2]. This notation is quite natural: For instance, if one wants to define the `while` statement in C language that has the form:

`while` ($expression$) $statement$;
one just writes the following:
$statement \rightarrow$ `while` ($expression$) $statement$;
This is called a *production* or a *rule*. Before the arrow, the variable $statement$ is called a nonterminal. The arrow means "can be seen as". On the right side of the arrow, there is the result of the production: the keyword `while`, the parentheses and the semicolon that are all tokens passed from the lexer. $expression$ is another nonterminal that has to be defined in the parser.

To define a grammar, in addition to the tokens (also known as terminal symbols), the nonterminals and the productions, One needs to define a *start* symbol among the nonterminals. This symbol is the starting rule of the grammar.

1.4 Using yacc++

Yacc++ is a follow on to yacc and lex. Yacc++ is designed to provide an easier way to design both the lexer and the parser in the same file. This will not describe every new feature that has been added since lex and yacc. See the reference manual [4] for an in-depth look at yacc++. The following discusses how to define a grammar with yacc++.

1.4.1 Overview of yacc++

The lexer and the parser are both defined in the same file, so that only one file needs to be compiled with yacc++. Yacc++ is an object-oriented evolution of lex and yacc. With yacc++, comes the Language Objects Library, that provides some C++ classes and tools to handle objects. The declarations and definitions of the tokens and the rules are natural. The notation is very close to the BNF and is the same in the lexer and the parser: the arrow is replaced by a colon and a semicolon must be put at the end of the productions. Recursion can be avoided using '+' (that means one or more times) and '*' (that means zero, one or more times). Other useful symbols are '|' that means "or" and '?' for zero or one time (the '+', '*' and '?' are placed just after the concerned symbol). Also, yacc++ gets rid of left recursion by itself.

Yacc++ is object-oriented. Both the lexer and the parser are declared as objects in the Language Objects Library, as well as the symbol table. After parsing, the parser object contains the generated Abstract Syntax Tree (AST).

1.4.2 What is an Abstract Syntax Tree (AST)?

While parsing, the parser builds a *parse tree*, according to the production rules. The start symbol is the root of the tree. An AST is a part of the parse tree. To construct such a tree, each parsed nonterminal one wants to see in the tree must be declared in a *construct* declaration. Only the classes of the *construct* declarations belong to the AST. After the parsing, the AST can be accessed for the information needed. As the information remains in the AST, one doesn't have to insert code in the parser any more to store this information somewhere else or process it while parsing. However, this possibility still exists and one can put some code that will be executed while parsing. For instance, one can tell the parser to print something on the screen, each time a particular token is encountered. Nevertheless, to be able to access the information after parsing is a great advantage, mainly because one has access to all the information the parser has been able to gather from the input.

As each *construct* declaration will be transformed into a C++ class by yacc++, one can customize the classes, adding as many fields as one wants in each class. These fields are initialized when one grammar symbol corresponding to the class is parsed.

1.4.3 Choice of the classes belonging to the AST

At first sight, the choice of the classes seems to be quite easy, but there are a few things to care about: One needs to have a root, to be able to access the AST and not to separate it in several parts. One also has to make sure that each class of the AST is a direct production of another class (except for the root). Considering this, one can choose and build the classes.

1.4.4 Example of an AST built by AutoMap

See figure 1.1 on page 7. This example shows the AST constructed by AutoMap, when the input is the following file:

```
/*~ AutoMap_Begin */

struct event {
  int pid;                /* -1 indicates null */
  int row_src, col_src;   /* range [1..n], [1..m] */
  int row_dst, col_dst;   /* range [0..n], [0..m+1] */
  float rate;
  float priority;
```

```
};

typedef struct event event_struct /*~ AutoMap_CTpUsed */;
/*~ AutoMap_End */
```

This AST is constructed according to the grammar of AutoMap. For more details on the structure of the AST and the AutoMap grammar, please see chapter 2. However, one can see that some tokens are missing, such as the keyword `struct`, the semicolons or the braces. In the AST, one keeps only the relevant information.

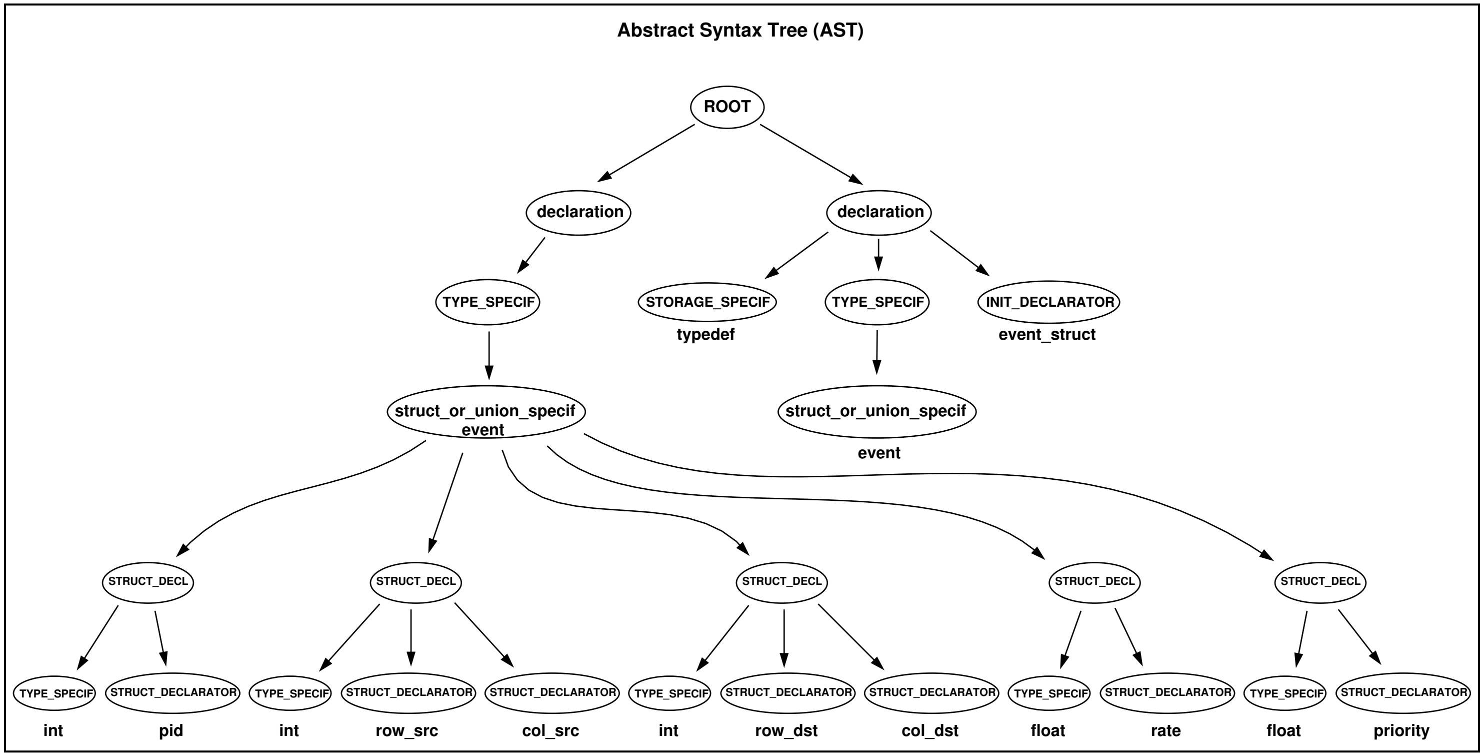

Figure 1.1: AST

1.5 MPI Datatype Issues

In the case of structures, there are possible interactions between MPI and the compiler [1]. These interactions can affect the way a compiler does padding between one structure and the next.

If you are sending more than one structure, this can be an issue. There are two possible ways to deal with this. One way is to assume default padding and use the CASE 1 scenerio shown below to create the structure typemaps. The other way is to explicitly include the upper bound with `MPI_UB`. This is CASE 2 below.

It is useful to assume CASE 1, but have a test that would warn you if things did not match up. The test case could create both datatypes, with and without `MPI_UB`, get their extents, and compare. The current version of AutoMap implements CASE 1, but a future version will provide both options, with a test to ensure correctness if CASE 1 is used.

Typically `MPI_UB` isn't needed because the compiler has a default padding algorithm: add padding to move the address to the next object boundary, where the object is the largest intrinsic data type used. LAM [3] does this, and vendor libraries are expected to do this.

The *safer use* escape hatch is to handle these cases where compilers provide switches to modify the default padding rules. It also helps public/portable MPIs not have to special cases for some potential vendor's complex padding rules. In these case, the library doesn't know how many padding bytes to add. For these corner-cases, users are advised to let the compiler compute the padding by taking the address of the next structure element in an array, and using that to let `MPI_UB` force the padding. CASE 2 may be less optimal on some MPIs.

The padding is local; the wire protocol packs the data, removing all holes. The only requirement is that the typemaps match. When the data is unpacked, this is done at the destination, using it's own datatype to recreate the holes as required locally.

Extent is important on the local side, in order to determine how to access the data while packing and unpacking. The basic problem is how to determine where the following element is, given the current one (the case of sending $N > 1$ elements. The rule is: displacement += extent. CASE 1 versus CASE 2 shows up in telling MPI how to pack/unpack, not when the data goes on the wire.

```
Example:
```

[1]The authors acknowledge the work of Raja Daoud on this section.

```
struct foo {
int i;
char c;
};

struct foo data[2];
char *buf;

MPI_Type_struct(...., &dtype);
MPI_Type_commit(&dtype);

buf = (char *) data;
MPI_Send(buf, 2, dtype, ...);
```

After packing the first structure in *buf* (the first 5 bytes), where does the second one start? With CASE 1, a correct MPI implementation, and default padding rules (assuming 32-bit integers), it should be at buf[8]. If the user makes the compiler change its layout rules, say always pad to 16-byte boundaries, the data would be at buf[16], but MPI would still assume regular padding and use buf[8]. With CASE 2, *dtype* would have been constructed with `MPI_UB` at 16, as given by the compiler layout via `&data[1]`.

```
CASE 1 (example 3.34, MPI 1.1 standard page 79):
The structure to be converted is the following:


struct Partstruct
   {
     int class;    /* particle class */
     double d[6];  /* particle coordinates */
     char   b[7];  /* additional information */
   };


First the name of the MPI data structure is declared:

MPI_Datatype Particletype;

Then the data types of each of the fields of the structure
must be given.  This is followed by the number of items of
each field.

MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
int blocklen[3] = {1, 6, 7};

/* declare an array to contain the displacements */
MPI_Aint   displ[3];

/* base holds the base address of the structure       */
int        base;
```

```
/* compute displacements of structure components */
MPI_Address(particle, disp);
MPI_Address(particle[0].d, disp+1);
MPI_Address(particle[0].b, disp+2);
base = disp[0];
for(i = 0; i < 3; i++) disp[i] -= base;

The MPI type is then declared and committed.

MPI_Type_struct(3, blocklen, disp, type, &Particletype);
MPI_Type_commit(&Particletype);



--------------------------------------------
However, the same example says "if the compiler does padding in
mysterious ways the following *may* be safer"

CASE 2:
- same structure but declare the type as follows:


MPI_Datatype type[4] = {MPI_INT, MPI_DOUBLE, MPI_CHAR, MPI_UB};
int blocklen[4] = {1, 6, 7, 1};


/* declare an array to contain the displacements */
MPI_Aint   displ[4];

/* base holds the base address of the structure       */
int        base;

/* compute displacements of structure components */
MPI_Address(particle, disp);
MPI_Address(particle[0].d, disp+1);
MPI_Address(particle[0].b, disp+2);
MPI_Address(particle+1, disp+3);
base = disp[0];
for(i = 0; i < 4; i++) disp[i] -= base;

The MPI type is then declared and committed.

MPI_Type_struct(4, blocklen, disp, type, &Particletype);
MPI_Type_commit(&Particletype);
```

Chapter 2

AutoMap version 1.1 Design and Implementation

2.1 Introduction and Overview

AutoMap recognizes the structures to be converted to MPI types by marks or directives in the C code. In order to be able to read a user's code directly, the directives were designed as modified C comments. The modification consists of a '~' just after the usual beginning of a comment '/*'. Additionally, this is followed by an `AUTOMAP_BEGIN` for the start directive or an `AUTOMAP_END` for the end directive. Thus a structure is surrounded by:

```
/*~ AUTOMAP_BEGIN */

/*~ AUTOMAP_END */
```

AutoMap is designed to be used as a stand-alone utility; but it is also designed to work with AutoLink, the dynamic data structures MPI library. Since an MPI-message can only be one array of instances of a certain datatype, to efficiently send a linked datastructure all instances of the same datatype need to be in a single large array. AutoLink uses three different ways of handling memory-objects to accomplish this:

1. *single:* object is created independently and is referred to by it's address.

2. *string*: objects are part of an array and are referred to only by the first instance in the array.

3. *array*: objects are created as part of an array but can be referred to individually.

It is the user who determines in which of the above ways instances of his datatypes are handled. This information is provided to AutoMap through directives in the user's code. The directives corresponding to each are:

type	directive
single	`/*~ AutoMap_CTpUsed */`
string	`/*~ AutoMap_STpUsed */`
array	`/*~ AutoMap_HTpUsed */`

This is exemplified in the following structure where the begin and end and type is indicated. Note that the directive comes *before* the semicolon.

```
/*~ AutoMap_Begin */

struct event {
  int pid;                /* -1 indicates null */
  int row_src, col_src;   /* range [1..n], [1..m] */
  int row_dst, col_dst;   /* range [0..n], [0..m+1] */
  float rate;
  float priority;
};

typedef struct event event_struct /*~ AutoMap_CTpUsed */;
/*~ AutoMap_End */
```

Another feature added to AutoMap was the possibility to use options ('-c' to specify the use of C language) that allow further improvements like the implementation of other grammars for other languages such as C++, Fortran-77 and Fortran-90.
But the most important: The automatic generation of MPI data types. The AutoMap output includes a file: `mpitypes.c`, that contains the code to create the MPI data types corresponding to the structures declared as CTp, STp or HTp. This part of the output can be used without AutoLink: To send a basic structure with MPI, the file `mpitypes.c` can be used to generate the MPI data types. Thus there is also a stand-alone version of AutoMap, that outputs only one file (`mpitypes.c`) and uses only the CTp type.

2.2 Design of AutoMap

AutoMap actually contains two grammars. One grammar recognizes the *start* directive. The other grammar reads and converts the recognized structs. There are three source files: the two grammars—`AutoStart.yxx` and `AutoMap.yxx`— and the main program `AutoMap_main.yxx`. Logically, AutoMap contains two main parts: The parsing part (grammar) that reads the input file, and the main routine that makes use of the information collected by the first part. The figure 2.1 on page 13 shows how AutoMap is compiled into a single executable file, with the `makefile` [appendix B].

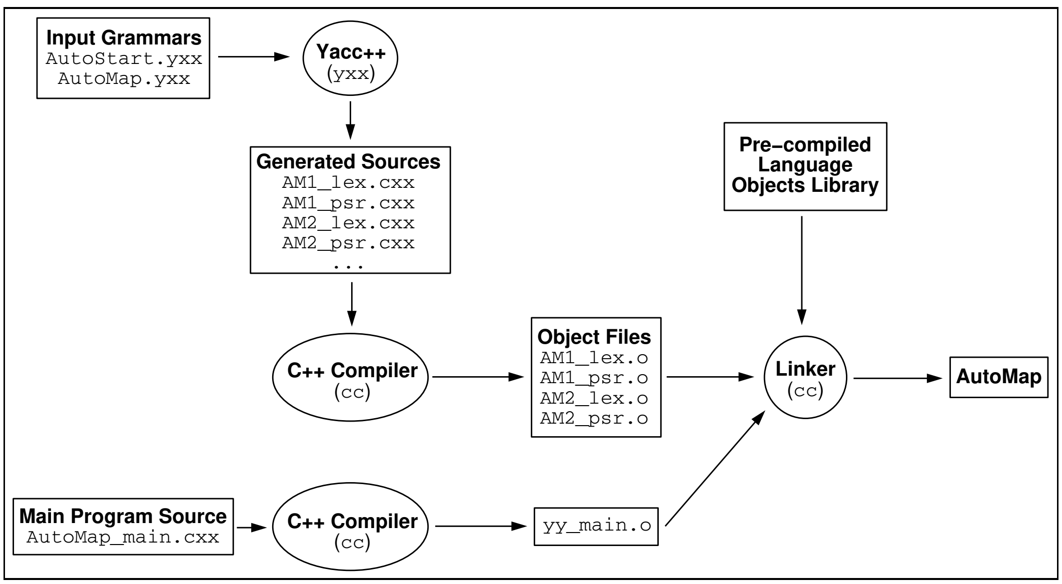

Figure 2.1: Compilation of AutoMap

The first grammar (`AutoStart.yxx`) detect the begin mark, then stops and give the stream to the main grammar that builds the AST. Once the parse is completed, the AST is given to the main program, through the parser object. Using the AST, the main program outputs the files. This is how AutoMap works. See figure 2.2 on page 15. As mentioned previously, there are two versions of AutoMap: one stand-alone and one to go with AutoLink. The grammars are the same and the difference between the two versions are made with conditional compilation directives inside the main program. However, as such directives can't be used in a yacc++ file (except in the global declaration), this was not enough and the file that contains the main grammar (`AutoMap.yxx`) has two versions: `AutoMap.yxx` for the version to work with AutoLink and `AutoMap_sta.yxx` for the stand-alone version. The macro definition that tells the version (`#define STANDALONE`) is defined in the file `AutoMap_sta.yxx`, so the two other files (`AutoStart.yxx` and `AutoMap_main.cxx`) are exactly the same in both versions.

2.3 The First AutoMap Grammar

The first AutoMap grammar is `AutoStart.yxx`. The complete file is in appendix D. This grammar is designed to detect the "begin" mark in the user's code. AutoStart can read almost everything before this mark in the code (even if it is not C code), and stop at the mark to give the stream to AutoMap.

2.3.1 The lexer

Here is the code defining this lexer:

```
lexer :: NAME AM1_lexer; // lexer declarations and definitions

local {
        static int to_quit=0;
}

                // token list
token           IDENTIFIER
                ;

                // Tokens to be ignored
discard token   COMMENT  ANYCHAR
                ;


// Token definitions

IDENTIFIER   :  (("a" .. "z" | "A" .. "Z" | "_")
                ("a" .. "z" | "A" .. "Z" | "0" .. "9" | "_")*)
             ;

COMMENT      :   "/*" ( "~" (@)* AutoMap_stmt ( (@)* "*" )+ "/"
                      |   ( (@)* "*" )+ "/" )
             ;
                { if (to_quit) yy_lex_quit(); }

AutoMap_stmt :   "AutoMap_"
                ( "Begin" { to_quit=1; }
                | @ )
```

```
              ;

ANYCHAR       :    @ | "/"
              ;
```

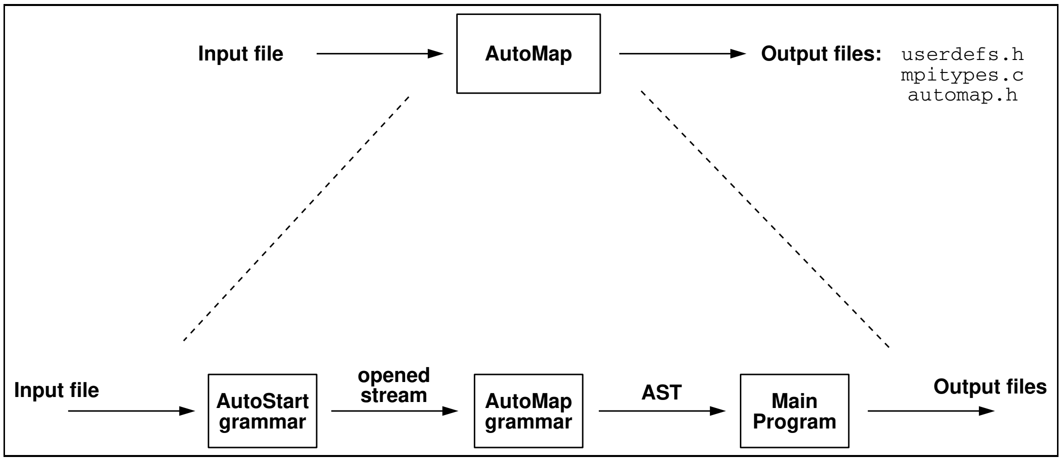

Figure 2.2: General design of AutoMap

Let's focus on the token definitions: The rule for an identifier means it can begin with any small or capital letter or an underscore. From the second character on, there can be a number. The token `IDENTIFIER` is then reduced as soon as the lexer reads something else than a letter, a number or a '_'.

`COMMENT` and `ANYCHAR` are declared as "discard token", that means they are not passed to the parser. The token `ANYCHAR` uses the symbol '`@`' that replaces every character that is not expected by the lexer. The lexer can have different states: it can be reading an identifier for instance or just waiting for a new token to begin, etc. If the lexer is expecting a new token, it can be any of the three tokens declared: `IDENTIFIER`, `COMMENT` or `ANYCHAR`. If the lexer reads a letter or '_', its state changes to "reading an identifier". If it reads '/', then it's the token `COMMENT` or `ANYCHAR`, depending if the following character is a '*'. For any other character, the token `ANYCHAR` will be reduced.

The '/' added in the `ANYCHAR` rule aims to prevent an error when the lexer reads a '/' not followed by a '*'. The `@` symbol is also used inside the token `COMMENT`. Its meaning remains the same: "every character not expected".

Yet, this grammar doesn't allow use of comments in any way: If one puts a '~' just after "/*", then it has to be an AutoMap directive, and as soon as there is an '`A`' in this kind of comment, it has to be followed by "`utoMap_`" or an error will occur. A way to make this grammar more flexible would be to use a keyword or a token for the directives and then detect the mark in the parser. But, as the mark is inside a comment, which is a discard token, it is not passed to the parser. Hence, the only token passed to the parser is IDENTIFIER, which makes the parser rather simple.

When the begin mark is detected the variable to_quit is set to 1, by some *shift action code* embedded inside the rule, with braces. Then, the lexer reduces the token COMMENT, executes the *reduce action code* associated to the token and quits (which makes the parser quit also). Notice the difference between shift and reduce action code: they are both embedded code in the grammar but the shift one is executed just after a *shift* inside a rule is performed (a shift correspond to a change of parser state, with no rule reduction), whereas the reduce one is executed only once the rule is reduced, and located after the semicolon ending the rule in the code.

2.3.2 The parser

The code for this parser is very small:

```
parser  :: NAME AM1_parser    // parser definitions
;

////////////////////////////////////////////////////////////////////////////////
// Beginning of the parser rules                                              //
////////////////////////////////////////////////////////////////////////////////

start   : (IDENTIFIER)*
        ;
```

The only production is the start symbol, that can be derived in any number of identifiers, until the lexer finds the begin mark and quits. The lexer stops on the first begin mark. To use several sections, with several “begin” and “end”, as the parsers are executed from the main program, it should be controlled by this main program. However, the use of several “AutoMap sections” in the user’s code, is not implemented yet: The main program would have to run the parsers several times which would produce several ASTs, to be grouped together.

2.4 The Second AutoMap grammar

The Second AutoMap grammar is AutoMap.yxx. The file AutoMap.yxx (in appendix E) gathers both the lexer and the parser of AutoMap. For the stand-alone version, it's the file `AutoMap_sta.yxx` (in appendix F). The only difference between the two versions is the absence of output to the file `userdefs.h` in the stand-alone version.

2.4.1 The lexer

This lexer can be divided into several parts: from the keyword and token declarations to the construct declaration and token definitions.

Definition of the keywords

```
// list of keywords
keyword         struct char int float void short long double signed unsigned
                const volatile typedef union enum auto register static extern
                ;
```

This declaration changes the way these words are passed from the lexer to the parser. Now, they are passed as keywords, and no longer as identifiers. The parser can make distinction between them.

The token and discard token declarations

```
// Token list
token           LBRACE  IDENTIFIER  INTEGER_CST  RBRACE  SEMI  COLON  LBRACK
                RBRACK  STAR  COMMA  EQUAL  LPAR  RPAR ;

                // Tokens to be ignored
discard token   WHITE_SPACE  COMMENT  define_stmt  include_stmt
                ;
```

The construct declaration

```
// construct declarations
construct    IDENTIFIER :: keyword

  base { public struct_base_class }

  constructor body {
    yy_symbol()->yy_sym_type(yy_symbol()->yy_sym_tkn_type());
  } ;
```

This declaration is needed, if one wants the lexer to check if an identifier is a keyword or not, before passing it to the parser. Actually, yacc++ generates automatically the reduce action code needed to perform this checking.

The token definitions

As the complete file is in appendix E, this looks at the main tokens such as identifiers and comments.

```
IDENTIFIER   :  (("a" .. "z" | "A" .. "Z" | "_")
                ("a" .. "z" | "A" .. "Z" | "0" .. "9" | "_")*)
                      { ident_ptr = new char[yy_lex_len() + 1];
                        memcpy(ident_ptr, yy_lex_token(), yy_lex_len());
                        ident_ptr[yy_lex_len()] = '\0';
                        if (read_struct == 1) { read_struct=0;
                          struct_name = new char[yy_lex_len()+1];
                          memcpy(struct_name, yy_lex_token(), yy_lex_len());
                          struct_name[yy_lex_len()] = '\0'; }
                        fprintf(header_ptr, ident_ptr); }
             ;
```

The rule is the same as in AutoStart. The shift action code added to the identifier definition aims to memorize the last identifier name in a global variable, so that the parser can access directly this name, via the variable. This shift action code is located so that it actually works like a reduce action code. However, as the token `IDENTIFIER` has a construct declaration, yacc++ generates automatically some reduce action code and we can't overwrite it. That's why shift action code is used. The identifier is also memorized in another variable (`struct_name`) if it is a structure name. The variable `read_struct` is used by the parser to tell the lexer when it is such a case. The token `INTEGER_CST` also use a global variable to store its value as a character string, that is particularly useful for the array sizes.

As to the `fprintf` command, it is used to write the token to the file `userdefs.h`. In this version, each token, except the comments are copied to this file. The result is a copy (without the comments) of the part of the user code located between the "begin" and "end" marks. This way, the declarations the user wants to use with AutoLink, are automatically included in `autolink.c`, via the file `userdefs.h`. The comments and AutoMap directives are not included in the output file.

```
WHITE_SPACE  :   (" "             // space
             |    "\t"            // tab
             |    "\n"            // newline
                                  {  ++yy_cur_lineno();  }
                  )+
             ;
        { token_ptr = new char[yy_lex_len() + 1];
          memcpy(token_ptr, yy_lex_token(), yy_lex_len());
          token_ptr[yy_lex_len()] = '\0';
          fprintf(header_ptr, token_ptr); }

COMMENT      : "/*" ( "~" (@)* AutoMap_stmt ( (@)* "*" )+ "/"
```

```
                        |    ( (@)* "*" )+ "/" )
              ;
                { if (to_quit_psr) {
                     yy_this_psr_obj->yy_psr_quit();
                     yy_lex_rdc() = yy_eof_; }
                 }


AutoMap_stmt :   "AutoMap_" ( "CTpUsed"  { ctpused=1; }
                            | "STpUsed"  { atpused=1; }
                            | "HTpUsed"  { htpused=1; }
                            | "End"      { to_quit_psr=1; }
                            | @ )
              ;

define_stmt  :   "#define" @* "\n" { ++yy_cur_lineno(); }
              ;
         { token_ptr = new char[yy_lex_len() + 1];
           memcpy(token_ptr, yy_lex_token(), yy_lex_len());
           token_ptr[yy_lex_len()] = '\0';
           fprintf(header_ptr, token_ptr); }

include_stmt :   "#include" @* "\n" { ++yy_cur_lineno(); }
              ;
         { token_ptr = new char[yy_lex_len() + 1];
           memcpy(token_ptr, yy_lex_token(), yy_lex_len());
           token_ptr[yy_lex_len()] = '\0';
           fprintf(header_ptr, token_ptr); }
```

These are the discard tokens. They are not passed to the parser, unless a reduce function is used (`yy_lex_rdc()`), like in the reduce action code associated to `COMMENT`. The `AutoMap_stmt` uses global variables to tell the parser which types are going to be used as XTp.

2.4.2 The parser

In the parser section, there are all the construct declarations for the AST as well as the rules.

The global and union declarations

The union declarations are absolutely necessary to access the AST. Each class must be accessed with the right type, either to go further in the tree or to access a particular field of a class.

```
union    {
           base_ptr               as_base_ptr;
           decl_file_ptr          as_decl_file_ptr;
           declaration_ptr        as_declaration_ptr;
```

```
          type_specif_ptr       as_specif_ptr;
          typedef_name_ptr      as_typedef_name_ptr;
          typedef_ptr           as_typedef_ptr;
          struct_ptr            as_struct_ptr;
          decl_ptr              as_decl_ptr;
          declarator_ptr        as_declarator_ptr;
        }
```

The construct declarations

```
//construct declaration for base class

base construct struct_base_class ::

  member {
    public:
        virtual yy_sym_ptr yy_symbol()          { return(NULL); }
        virtual const char *display_name()      { return(NULL); }
        virtual int        type()               { return(0); }
  };
```

The construct declarations form a very important part: They enable the parser to build the AST. In fact, these declarations are the classes of the AST. The parser needs first a base class to be able to build the AST. This base class must fit all the other classes because all the classes must derive (in the C++ sense) from this base class if one wants to include them in the AST.
In a usual construct declaration, there are several fields:

```
construct struct_decl_file :: all

base { public struct_base_class }

constructor body {
        cout << "AST constructed" << endl;
}

constructor initializer {
  num_structures (num_struct),
  num_typedefs  (num_typedef),
  num_unions    (num_union),
  num_typedef_structs   (num_typedef_struct),
  num_typedef_unions    (num_typedef_union) }

member {
  public:
    int         num_structures;
    int         num_typedefs;
    int         num_unions;
    int         num_typedef_structs;
    int         num_typedef_unions;
};
```

First, a class can derive from another: this is specified by the base field. The constructor body is code to be executed each time an instance of this class is reduced by the parser. In this case, The `struct_decl_file` is the start symbol of the grammar, so it's the root of the AST and it will be reduced only once (and then print on the screen the message "AST constructed"). The constructor initializer groups some members of the class, with initializing values that are affected once the instance is reduced. At last, the member field groups all the members of the class, except of course the pointers to other classes, that depend on the parse. Let's see the construct declaration for the declaration class:

```
construct declaration :: all

base { public struct_base_class }

constructor initializer {
  decl_type (class_type) }

constructor body {
  switch (class_type) {
    case STRUCT_TYPE           : ++num_struct; break;
    case UNION_TYPE            : ++num_union; break;
    case TYPEDEF_STRUCT_TYPE   : ++num_typedef_struct; break;
    case TYPEDEF_STRUCT_TYPE2 : ++num_typedef_struct; ++num_struct; break;
    case TYPEDEF_UNION_TYPE    : ++num_typedef_union; break;
    case TYPEDEF_UNION_TYPE2   : ++num_typedef_union; ++num_union; break;
    case TYPEDEF_TYPE          : ++num_typedef; break;
  }
}

member {
  public:
    int         decl_type;
    int         type() { return(decl_type); }
};
```

This class is the base class after the root. It fits all the C declarations. All the declarations have the same class. A type field in the class stores the type of the declaration. The types are names defined as integers in the global declaration:

```
#define STRUCT_TYPE             1
#define UNION_TYPE              2
#define TYPEDEF_STRUCT_TYPE     3
#define TYPEDEF_STRUCT_TYPE2    4
#define TYPEDEF_UNION_TYPE      5
#define TYPEDEF_UNION_TYPE2     6
#define TYPEDEF_TYPE            7
```

`STRUCT_TYPE` is for the simple structure declarations, without typedef. `TYPEDEF_STRUCT_TYPE` is used for a typedef on a structure, without the structure definition, as this one:

```
typedef struct event event_struct;
```

TYPEDEF_STRUCT_TYPE2 is used for a typedef on a structure with its definition in the same declaration, as:

```
typedef struct {
  int pid;               /* -1 indicates null */
  int row_src, col_src;  /* range [1..n], [1..m] */
  int row_dst, col_dst;  /* range [0..n], [0..m+1] */
  float rate;
  float priority;
} event_struct;
```

It is the same for the union types, as union and structure declaration are similar. At last, the TYPEDEF_TYPE is used for typedefs that are not on structures or unions. See appendix E for all the construct declarations.

The parser rules

The figure 2.3 on page 23 shows the organization of the classes with their fields, according to the grammar described in the parser rules.

The parser rules are very close to the official rules [8, page 210-222], except for the token typedef_name, the constant expressions and the initializers. The typedef name is normally seen as a type specifier. The problem is that there can be several type specifiers before a declarator (that can be an identifier naming a variable). In this case, the parser can't differentiate between a typedef name (that is an identifier) and the declarator. To implement the C grammar, the advice is to make the typedef_name a terminal symbol, instead of an identifier. To make the parser to read and reduce the typedef names, the grammar was changed. The official grammar for a declaration is equivalent to this:

```
declaration  :  ( STORAGE_SPECIF | TYPE_SPECIF | TYPE_QUALIF )*
                ( INIT_DECLARATOR  (COMMA  INIT_DECLARATOR)* )?
                SEMI
             ;
```

To prevent the reduction of several typedef names or the reduction of a typedef name after a type specifier, the modified rule is:

```
declaration  :  ( STORAGE_SPECIF | TYPE_QUALIF )*
                ( TYPEDEF_NAME | TYPE_SPECIF )
                ( STORAGE_SPECIF | TYPE_QUALIF | TYPE_SPECIF )*
                ( INIT_DECLARATOR  (COMMA  INIT_DECLARATOR)* )?
                SEMI
             ;
```

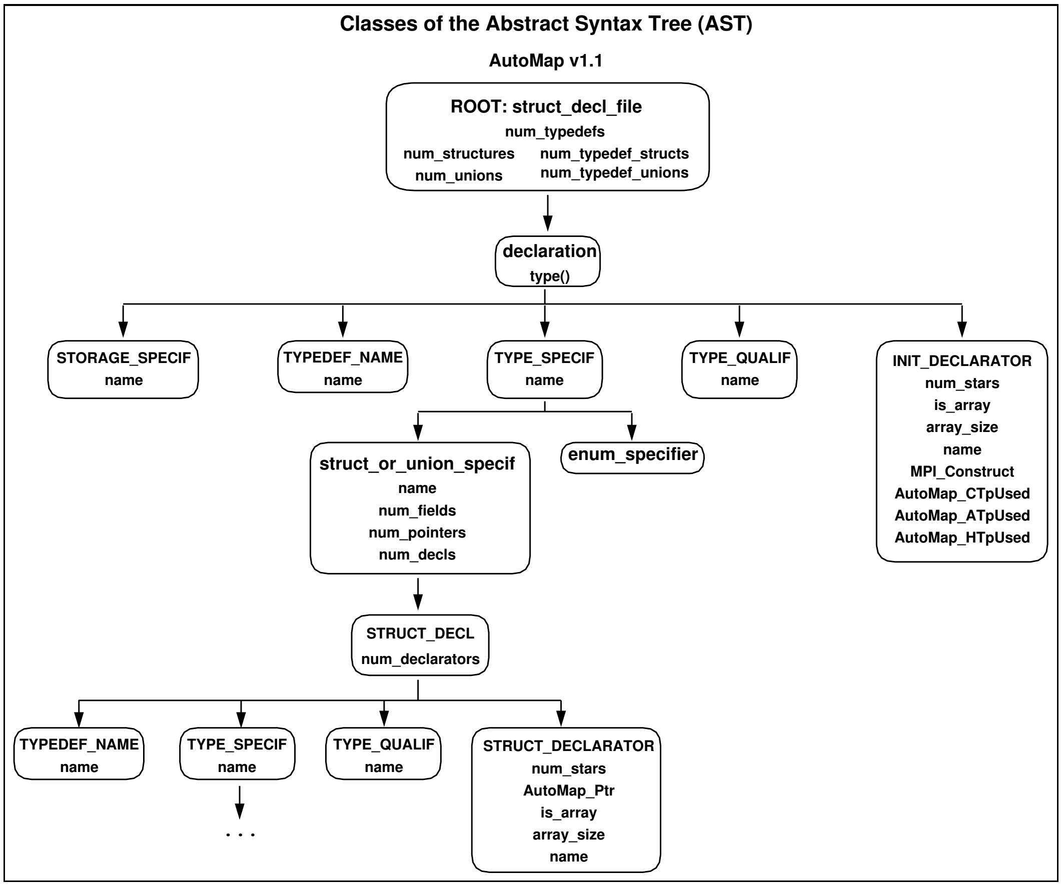

Figure 2.3: AST Classes

The constant expressions are simply limited to integer constants or identifiers (used as macros), instead of conditional expressions. As to the initializers, they use the following rule:

```
INITIALIZER  :  CST_EXPR
             |  LBRACE  INITIALIZER ( COMMA  INITIALIZER )*  COMMA ? RBRACE
             ;
```

instead of this one:

```
INITIALIZER  :  assignment_expression
             |  LBRACE  INITIALIZER ( COMMA  INITIALIZER )*  COMMA ? RBRACE
             ;
```

Here, the assignment expression is limited to a constant expression. These two restrictions find a justification in the fact that AutoMap needs only the definition of the types and not the variable declarations or initializations. If a declaration doesn't fit these requirements, it must be placed outside the AutoMap marks.

However, some more restrictions exist: they are not due to the grammar but to the use of global variables to store, for instance, the name of a structure. To use nested structure declarations, like this one:

```
struct point {
  int x;
  int y;
};

struct rect {
  struct point pt1;
  struct point pt2;
};
```

The user has to use typedef names because this is not handled properly by AutoMap yet, and it has to be written this way:

```
typedef struct point {
  int x;
  int y;
} pt_struct;

struct rect {
  pt_struct pt1;
  pt_struct pt2;
};
```

It's the same for unions, or to put unions declarations inside structures, as well as structures inside unions.

2.5 The AutoMap Main Program

The main routine is in a separate file: `AutoMap_main.cxx` in appendix G. With yacc++, one doesn't necessarily need to use a main routine, for simple lexers and parsers. Since AutoMap uses an AST, a main program was written to handle it. Also, to use two grammars in a single final executable file, one needs a main routine to control it. In this case, one has to declare all the objects one wants to use: lexers, parsers, etc. We also have to execute the parsers in this routine. It has to be in a separate file because it is written in C++. Obviously, the first advantage of this is the possibility to execute several parsers and to execute them several times for instance (eventually on different inputs).

Here, the main program first runs the AutoStart grammar, then the AutoMap one that builds the AST. To output the files, the main routine needs to access the AST. The next section explains how to access the AST. In the following ones, the Implementation of `AutoMap_main.cxx` will be detailed.

2.5.1 Accessing the AST

The AST is built by the AutoMap parser, so it has to be accessed via this parser object, once the parse is completed. To do so one needs to learn how to deal with the Language Object Library. Here is the root of the AST:

```
// root of the AST
root = yy_parser->yy_psr_ref(1).as_decl_file_ptr;
```

To access the AST classes, one has to use the union declarations. So, if there are enough declarations, one can access a class as a base class:

```
root->yy_operand(0).as_base_ptr
```

or as a specific class:

```
root->yy_operand(0).as_declaration_ptr
```

If a class is customized and one needs to access one of its field which is not a base class field, then one must access it as a specific class, using the union declarations. That means one has to know what is the type of this class. In this example, one accesses a declaration class: To go further in the AST and access the type specifier linked to it, the declaration class has to be accessed as a specific class.

2.5.2 Implementation overview

Preprocessor directives are used for conditional compilation in the main program code to compile the two different versions (stand-alone and with AutoLink). The file is composed of several subroutines that are declared before the main routine. They make the main shorter, more simple and easier to understand. The complete file is in appendix G.

2.5.3 The output files

AutoMap can be run as a standalone utility or as a companion to AutoLink. Before going further, let's have a closer look at the global design of AutoLink and AutoMap, and how they work together: AutoMap provides AutoLink with the generation of MPI data types and some information on these data types (like the fields that are pointers and the type of the data pointed). Before using AutoLink, AutoMap first has to be run on the file containing the structure declarations used. The AutoMap output is needed to initialize and compile AutoLink before using it. The figure 2.4 from [6] shows how to use AutoMap and AutoLink before compiling an application. AutoMap actually outputs three files: `userdefs.h`,

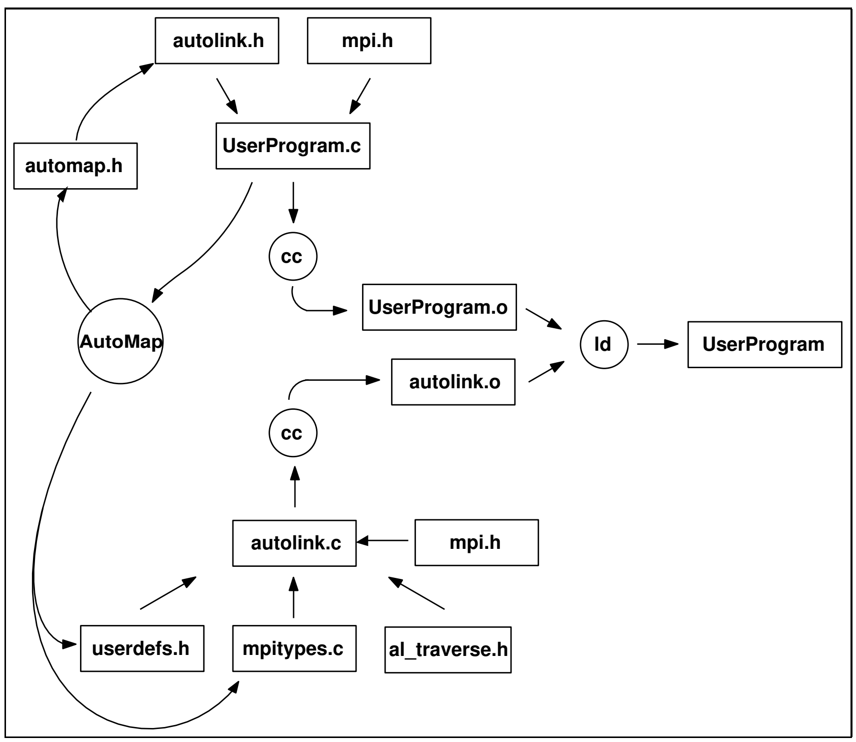

Figure 2.4: Compile with AutoMap and AutoLink

`mpitypes.c` and `automap.h`. The file `automap.h` has to be included in `autolink.h`, that is itself included in the user's code, just like the header file of a usual library. However, `autolink.c` has to be compiled with the two other files output from AutoMap. One (`mpitypes.c`) contains the code to generate the MPI data types, the other (`userdefs.h`) holds needed information on these types. All this compilation process can be automated with an appropriate makefile. For more information on AutoLink and how to use it, refer to [6].

Let's run AutoMap on the following file to better see the structure of the output files.

```
...
/*~ AutoMap_Begin */

struct Treeinfo {
  int Depth;
  int Stop;
  int NodeType;   /* 0 means data; 1 means a function; -1 means not assigned */
  int funct;      /* integer indexes into the function table                */
  int TypeData;   /* 0 means BiggerReal; 1 means array data; -1 means not
                     assigned                                               */
  int Index;      /* indexes into array data                                */
  int Nchildren;
  float Data;     /* for putting BiggerReals into   */
};

typedef struct Treeinfo TreeInfo;
typedef struct tnode  *Treeptr;

struct tnode {
         TreeInfo   Info;
         Treeptr    Parent; /* pointer to node's parent */
         Treeptr    left;
         Treeptr    middle;
         Treeptr    right;
         Treeptr    FarRight;
                };

typedef struct tnode Treenode /*~ AutoMap_CTpUsed */;

/*~ AutoMap_End */

TreeInfo *PInode;
...
```

For more complete examples, see the test chapter.

userdefs.h

This file contains vital information for AutoLink. Part of it comes from the answers to the AutoLink configuration routine, but the rest is automatically extracted from the AST. In this example, no information comes from the configuration questions.

```
#ifndef USERDEF
#define USERDEF
```

```
/* TYPES */


struct Treeinfo {
  int Depth;
  int Stop;
  int NodeType;
  int funct;
  int TypeData;
  int Index;
  int Nchildren;
  float Data;
};

typedef struct Treeinfo TreeInfo;
typedef struct tnode  *Treeptr;

struct tnode {
         TreeInfo   Info;
         Treeptr    Parent;
         Treeptr    left;
         Treeptr    middle;
         Treeptr    right;
         Treeptr    FarRight;
                };

typedef struct tnode Treenode ;



/* DEFINES */

#define MaxFields       50
#define CTpUsed         1
#define STpUsed         0
#define HTpUsed         0

typedef Treenode        CTp0;   /* datatype number 0 */

#define CT0

/* Treenode */
#define PtrFld00        Parent
#define PtrTp00         0
#define PtrFld01        left
#define PtrTp01         0
#define PtrFld02        middle
#define PtrTp02         0
#define PtrFld03        right
#define PtrTp03         0
#define PtrFld04        FarRight
#define PtrTp04         0

#define BSize0          100

int BSize[CTpUsed+STpUsed+HTpUsed+1] = { BSize0, 0 };
```

```
int BSizeb[STpUsed+HTpUsed+1] = { 0 };

#endif
```

The TYPES section is a copy of the code located between the “begin” and “end” marks, without the comments. The DEFINES section provides AutoLink with the configuration informations:

- Number of CTp, STp and HTp.
- typedefs on these types.
- For each XTp, a list of macros on the fields that are pointers to XTp: the name of the field and its type.
- Two arrays of block sizes.

For more details on how AutoLink uses this information, see [6].

mpitypes.c

This file contains the code to create the MPI data types corresponding to the structures used as XTp:

```
void Build_MPI_Types()
{
CTp0 CType0;

MPI_Aint        disp[MaxFields];
MPI_Datatype    type[MaxFields];
int             blocklen[MaxFields];
int             base;
int             i;

TreeInfo        TpDef0;
MPI_Datatype    AutoMap_TreeInfo;

/* TreeInfo */
MPI_Address(&TpDef0.Depth,      &disp[0]);
MPI_Address(&TpDef0.Stop,       &disp[1]);
MPI_Address(&TpDef0.NodeType,   &disp[2]);
MPI_Address(&TpDef0.funct,      &disp[3]);
MPI_Address(&TpDef0.TypeData,   &disp[4]);
MPI_Address(&TpDef0.Index,      &disp[5]);
MPI_Address(&TpDef0.Nchildren,  &disp[6]);
MPI_Address(&TpDef0.Data,       &disp[7]);
type[0] = MPI_INT;
type[1] = MPI_INT;
type[2] = MPI_INT;
type[3] = MPI_INT;
```

```
type[4] = MPI_INT;
type[5] = MPI_INT;
type[6] = MPI_INT;
type[7] = MPI_FLOAT;
blocklen[0] = 1;
blocklen[1] = 1;
blocklen[2] = 1;
blocklen[3] = 1;
blocklen[4] = 1;
blocklen[5] = 1;
blocklen[6] = 1;
blocklen[7] = 1;
base = disp[0]; for(i=0; i<8; i++) disp[i]-=base;
MPI_Type_struct(8, blocklen, disp, type, &AutoMap_TreeInfo);
MPI_Type_commit(&AutoMap_TreeInfo);

/* CTp0 */
MPI_Address(&CType0.Info,       &disp[0]);
MPI_Address(&CType0.Parent,     &disp[1]);
MPI_Address(&CType0.left,       &disp[2]);
MPI_Address(&CType0.middle,     &disp[3]);
MPI_Address(&CType0.right,      &disp[4]);
MPI_Address(&CType0.FarRight,   &disp[5]);
type[0] = AutoMap_TreeInfo;
type[1] = MPI_INT;
type[2] = MPI_INT;
type[3] = MPI_INT;
type[4] = MPI_INT;
type[5] = MPI_INT;
blocklen[0] = 1;
blocklen[1] = 1;
blocklen[2] = 1;
blocklen[3] = 1;
blocklen[4] = 1;
blocklen[5] = 1;
base = disp[0]; for(i=0; i<6; i++) disp[i]-=base;
MPI_Type_struct(6, blocklen, disp, type, &MPI_Types[0]);
MPI_Type_commit(&MPI_Types[0]);
}
```

Note that the structure `TreeInfo` is not declared as a XTp, but as this type is used in the `CTp0`, it is automatically created by AutoMap. This type creation is recursive: each time an unknown type is encountered, it is created before the current one.

automap.h

This file contains macro(s), for the user to refer to the types in order to use them with AutoLink.

```
/* AutoMap DEFINES */

#define Treenode_AutoNbr        0
```

2.5.4 The AutoMap main Routine

The AutoMap main routine starts with checking the options and declaring the objects for yacc++ and the Language Objects Library. Then it runs the AutoStart lexer and parser (`code = yy_psr(yy_parser1);`), gives the stream to the AutoMap lexer and parser, and runs them (`code = yy_psr(yy_parser);`). Once the parse is completed, the root of the AST is accessed via the parser object:

```
// root of the AST
root = yy_parser->yy_psr_ref(1).as_decl_file_ptr;
```

The section `correct num_stars and AutoMap_Ptr` updates the `num_star` field of the `STRUCT_DECLARATOR`s, according to the type of the `STRUCT_DECL`. Once the AST is correct, the output of the files begins: First `userdefs.h`, then `mpitypes.c` and `automap.h`.

In the section "`Check if all MPI_Datatypes are defined`", note that to create new MPI data types, one first needs to declare them. That's why this section falls into two loops: one for the declarations, the other for the definitions. Both subroutines used in these loops are recursive (`Types_declare` and `Types_construct`). After this, comes the type construction for the XTps.

In this type construction, one needs also to translate the basic C types in MPI datatypes: this is done by the `find_MPI_Types` subroutines. For a union, AutoMap uses the `MPI_BYTE` type, that doesn't necessarily work on heterogeneous architectures. As it is desired that MPI do the conversion, it is necessary to know the type of a data, and if it is a union one can only know it at run time: The solution is to ask the user to write a callback to be used by AutoLink to know the type of this data, and then send it separately. This is among the things to implement to achieve the goals of AutoLink.

2.5.5 The subroutines

Here is a list of all the subroutines, with comments:

```
void error_msg();

void help_msg();

declaration_ptr
find_declaration (decl_file_ptr obj_ptr, int obj_type, int obj_rank);
     // Return a pointer to the declaration object of type "obj_type"
     // and of rank "obj_rank" in the root object pointed by "obj_ptr"

type_specif_ptr find_specif (declaration_ptr obj_ptr, int obj_rank);
```

```
     // Return a pointer to the TYPE_SPECIF object of rank
     // obj_rank in the declaration pointed by obj_ptr

typedef_ptr find_typedef (declaration_ptr obj_ptr);
     // Return a pointer to the first INIT_DECLARATOR object
     // in the declaration pointed by obj_ptr

struct_ptr find_struct (declaration_ptr obj_ptr);
     // Return a pointer to the struct object contained in the
     // declaration pointed by obj_ptr

decl_ptr find_decl (struct_ptr obj_ptr, int obj_rank);
     // Return a pointer to the STRUCT_DECL object
     // which contains the declarator of rank "obj_rank"
     // in the struct_or_union object pointed by "obj_ptr"

declarator_ptr find_declarator (decl_ptr obj_ptr, int obj_rank);
     // Return a pointer to the struct_declarator object
     // of rank "obj_rank" in the struct_decl object
     // pointed by "obj_ptr"

decl_ptr find_decl (struct_ptr obj_ptr, int obj_rank, int obj_stars,
                    int automap_ptr);
     // Return a pointer to the STRUCT_DECL object
     // which contains the declarator of rank "obj_rank" having "obj_stars"
     // star(s)
     // in the struct_or_union object "obj_ptr" points to

declarator_ptr find_declarator (struct_ptr obj_ptr, int obj_rank);
     // Return a pointer to the declarator object
     // of rank "obj_rank" in the structure object pointed by "obj_ptr"

declarator_ptr find_declarator(struct_ptr obj_ptr, int obj_rank, int obj_stars,
                               int automap_ptr);
     // Return a pointer to the declarator object
     // of rank "obj_rank" having "obj_stars" star(s)
     // in the struct_or_union object pointed by "obj_ptr"

type_specif_ptr find_specif (decl_ptr obj_ptr, int obj_rank);
     // Return a pointer to the TYPE_SPECIF object of rank
     // obj_rank in the STRUCT_DECL pointed by obj_ptr

typedef_name_ptr find_typedef_name (decl_ptr obj_ptr);
     // Return a pointer to the TYPEDEF_NAME object used in the STRUCT_DECL
     // pointed by obj_ptr

struct_ptr find_struct (decl_file_ptr obj_ptr, declaration_ptr type_ptr);
     // Return a pointer to the structure object
     // which the typedef pointed by type_ptr refers to

declaration_ptr find_declaration_typedef (decl_file_ptr obj_ptr, int obj_rank,
                                          int obj_stars);
     // Return a pointer to the declaration object which contains
     // the typedef_struct of rank "obj_rank", having "obj_stars" star(s)

declaration_ptr find_declaration_struct (decl_file_ptr obj_ptr, int obj_rank);
     // Return a pointer to the declaration object which contains
```

```
     // the structure of rank "obj_rank"

declaration_ptr find_CTp (decl_file_ptr obj_ptr, int obj_rank);
     // Return a pointer to the declaration object
     // which is the CTp of rank "obj_rank"

declaration_ptr find_ATp (decl_file_ptr obj_ptr, int obj_rank);
     // Return a pointer to the declaration object
     // which is the ATp of rank "obj_rank"

declaration_ptr find_HTp (decl_file_ptr obj_ptr, int obj_rank);
     // Return a pointer to the declaration object
     // which is the HTp of rank "obj_rank"

declaration_ptr find_XTp (decl_file_ptr obj_ptr, int obj_rank);
     // Return a pointer to the declaration object
     // which is the XTp of rank "obj_rank"

declaration_ptr find_declaration_typedef (decl_file_ptr obj_ptr,
                                          decl_ptr struct_decl_ptr);
     // Return a pointer to the declaration object which contains the typedef
     // which is used in the structure declaration "struct_decl_ptr" points to

declaration_ptr find_declaration_typedef_struct (decl_file_ptr obj_ptr,
                                                 decl_ptr struct_decl_ptr);
     // Return a pointer to the declaration object which contains
     // the typedef_struct
     // which is used in the structure declaration "struct_decl_ptr" points to

int find_ctp (decl_file_ptr obj_ptr, decl_ptr struct_decl_ptr);
     // Find the CTp number
     // of the type used in the structure field "struct_decl_ptr" points to

int find_atp (decl_file_ptr obj_ptr, decl_ptr struct_decl_ptr);
     // Find the ATp number
     // of the type used in the structure field "struct_decl_ptr" points to

int find_htp (decl_file_ptr obj_ptr, decl_ptr struct_decl_ptr);
     // Find the HTp number
     // of the type used in the structure field "struct_decl_ptr" points to

declaration_ptr find_declaration_typedef_union (decl_file_ptr obj_ptr,
                                                const char *type_name);
     // Return a pointer to the declaration object which contains
     // the typedef_union which typedef name is type_name

char *find_MPI_Type (decl_file_ptr obj_ptr, const char *type_name);
     // Find the MPI_Type corresponding to the C type which name is "type_name"

char *find_MPI_Type (decl_file_ptr obj_ptr, decl_ptr decl_p);
     // Find the MPI_Type corresponding to the C type
     // which is used in the struct_declaration pointed by decl_p

void AutoLink_Cfg(decl_file_ptr obj_ptr);
     // Configuration routine for AutoLink

void Types_construct(decl_file_ptr obj_ptr, int type, int num);
```

```
     // Write in mpitypes.c the code to create the MPI Types
     // type is 1 for CTp, 2 for ATp or 3 for HTp
     // num is the number of types to construct (num_ctp, num_atp...)

void Types_declare(decl_file_ptr obj_ptr, declaration_ptr declaration_p);
     // Write in mpitypes.c the code to declare the MPI Type
     // corresponding to the typedef declaration pointed by declaration_p

void Types_construct(decl_file_ptr obj_ptr, declaration_ptr declaration_p);
     // Write in mpitypes.c the code to create the MPI Types
     // corresponding to the typedef declaration pointed by declaration_p
```

Most of these subroutines are dedicated to help the programmer access AST. This way, one doesn't need to reorganize the data. One uses the AST. This makes it more flexible for further improvements or changes.

Chapter 3

AutoMap Uses and Test Suite

3.1 Uses

AutoMap can be used with AutoLink or as a stand-alone tool. There are actually two versions of AutoMap, but the grammars are exactly the same. In both cases, one has to tell AutoMap which structures one wants to send, that is to say which structures one wants AutoMap to "map". This is handled by the AutoMap directives put in the file to be read by AutoMap. To run AutoMap on a file, type: "`Automap file_name`". The name of the file is an argument. One can also use options: for instance '-c' for a C code file. The default option is C code (the only one available right now). Type "`AutoMap -help`" for more details.

3.1.1 Grammar restrictions

AutoMap does not read all the C Grammar. That's why one puts "begin" and "end" marks around the declarations you want to read. The marks are put into comments, not to be read by the compiler: `/*~ AutoMap_Begin */` and `/*~ AutoMap_End */` (see the following section for further details on the AutoMap directives and their format).

Within these marks, one can put typedef, structure and union declarations, as well as other declarations. One can also put some "`#define`" and "`#include`". For instance, if one used a macro as an array size, one can put it between the "begin" and "end". One can also put variable declarations that have no effect on the result. However, there are two restrictions detailed in 2.4.2: The constant expressions are limited to integer constants or identifiers and the assignment expressions are limited to constant expressions.

Lastly, if one wants to use nested structures, one has to use typedef names inside another structure declaration, as shown on page 24. However, the use of nested structures is supported by AutoMap and it generates automatically the MPI Types corresponding to a structure nested in a XTp, as a recursive process. See the example of the cube structure in the test suite of the stand-alone version.

3.1.2 AutoMap directives

The AutoMap directives must not be read by the compiler you run on your code, that's why they are put into comments. For C code, you type:
`/*~ AutoMap_CTpUsed */`
in front of the typedef that gives an alias on the structure you want to send. To know how to choose between the different declaration types (`CTp`, `HTp` or `STp`), look at the AutoLink User Manual [6]. However, to use AutoMap stand-alone, only the `CTp` type is available. Put the '~' sign just after the '`/*`' to make sure that AutoMap won't discard this comment. **Put this directive before the semicolon which ends the typedef declaration.**
As a XTp has to be a structure, the mark must be located in front of a typedef on this structure. This typedef can have the type `TYPEDEF_STRUCT_TYPE` or `TYPEDEF_STRUCT_TYPE2` as described in the type classification on page 21.
Here is an example of a section of a file that can be read by AutoMap:

```
...
/*~ AutoMap_Begin */

struct Treeinfo {
  int Depth;
  int Stop;
  int NodeType;   /* 0 means data; 1 means a function; -1 means not assigned */
  int funct;      /* integer indexes into the function table                 */
  int TypeData;   /* 0 means BiggerReal; 1 means array data; -1 means not
                     assigned                                                */
  int Index;      /* indexes into array data                                 */
  int Nchildren;
  float Data;     /* for putting BiggerReals into   */
};

typedef struct Treeinfo TreeInfo;
typedef struct tnode  *Treeptr;

struct tnode {
         TreeInfo   Info;
         Treeptr    Parent; /* pointer to node's parent */
         Treeptr    left;
         Treeptr    middle;
```

```
        Treeptr    right;
        Treeptr    FarRight;
              };

typedef struct tnode Treenode /*~ AutoMap_CTpUsed */;

/*~ AutoMap_End */

TreeInfo *PInode;
...
```

One can put the "end" mark after the declaration of `PInode`, but there is no point. In this example, if one wants to send an instance of `Treenode`, without sending each field separately, one needs to use AutoLink.
Here is another example, that uses with AutoMap stand-alone:

```
/*~ AutoMap_Begin */
typedef struct event {
  int pid;                /* -1 indicates null */
  int row_src, col_src;  /* range [1..n], [1..m] */
  int row_dst, col_dst;  /* range [0..n], [0..m+1] */
  float rate;
  float priority;
} event_struct  /*~ AutoMap_CTpUsed */;
/*~ AutoMap_End */
```

3.1.3 Stand-alone

With the stand-alone use, include the file `mpitypes.c` produced by AutoMap in your file(s). Then, use the MPI send and receive commands as desired.
Just before the main, type:

```
#include "mpitypes.c"
```

Then, after the `MPI_Init` call the `Build_MPI_Types()` function that is in `mpitypes.c`. In the `Send` and `Receive` commands, refer to the MPI Types built by putting the prefix "`AutoMap_`" before the name of the type. For instance, the MPI Type corresponding to the `event_struct` type is `AutoMap_event_struct`. To use different files for sending and receiving, include the file (`mpitypes.c`) produced by AutoMap in all the files concerned by the new MPI type(s).

3.1.4 With AutoLink

With AutoLink, also include the files produced by AutoMap in your own files. But this time there are three files: `userdefs.h`, `mpitypes.c` and `automap.h`.

For a linked structure it is necessary to use the send and receive routines from AutoLink. For further details on this, refer to the AutoLink documentation. There are now three files to include. `userdefs.h` and `mpitypes.c` are already included in `autolink.c`, see figure 2.4 on page 26. Include `automap.h` in `autolink.h` or directly in your code. This file is used to access the MPI types created in `Build_MPI_Types()`. This time, to make it convenient for AutoLink, the type are stored (actually, pointers to the types) in an array that is used by autolink to refer to the types. To refer to the type in the send and receive functions of AutoLink (`AL_Send()` and `AL_Recv()`), use a macro defined in `automap.h`: the name of the type followed by "`_AutoNbr`". This will give to AutoLink the reference of the MPI type in the array.

3.2 Test suite

3.2.1 Stand-alone version

A set of structures

One test program has several structures in it. There is one program to send the structures over MPI (`prog.c` in appendix H) to another program (`prog2.c` in appendix I) that runs on another node. This is accomplished with LAM (Local Area Multicomputer). With this MPI Implementation, the two programs can run on different architectures: MPI takes care of the data conversions. The structures of this test have been taken from [1, 7, 8], or inspired by some examples of these books. All the declarations are in the file `struct.h` in appendix J. Once AutoMap is run on this file, we have the file `mpitypes.c` (appendix K) and this output on the screen:

```
-- AutoMap for C files --
-- Stand-alone version --
First parse completed successfully.
AST constructed
Main parse completed successfully.
num_typedefs:   0
num_structures: 10
num_unions:     0
num_typedef_structs:    11
num_typedef_unions:     0
num_ctp: 6
num_atp: 0
num_htp: 0
0 Errors, 0 Warnings, Highest Severity 0
```

`prog` puts data in the structures and send the structures to `prog2` using the MPI data types defined in `mpitypes.c`. these two programs are actually two processes working in parallel, thanks to LAM. For details on how to use LAM, see [5]. Before receiving the structures, `prog2` has to declare some instances of the structures to be able to store them, and initializes them. `prog2` outputs two files: one before receiving the data and one after (called respectively `before` and `after`) to check if everything worked fine. Here is `before`:

```
pid val:        0
rate:   0.000000

maxiter=0
xmin=0.000000
ymin=0.000000
xmax=0.000000
ymax=0.000000
width=0
height=0
Quick lunch ?

class=0
d[0]=0.000000
d[1]=0.000000
d[2]=0.000000
d[3]=0.000000
d[4]=0.000000
d[5]=0.000000
b=aaaaaa

rect1:
  pt1: 0, 0
  pt2: 0, 0
rect2:
  pt1: 0, 0
  pt2: 0, 0

Depth: 0
Stop: 0
NodeType: 0
funct: 0
TypeData: 0
Index: 0
Nchildren: 0
Data: 0.000000
Parent: 0

fd: 0
ino: 0
name: struct.h
```

and here is `after`:

```
pid val:        15
```

```
rate:   23.670000

maxiter=26
xmin=-2.500000
ymin=-5.000000
xmax=12.534000
ymax=54.670000
width=10
height=7
Hello World !

class=2
d[0]=1.200000
d[1]=2.200000
d[2]=3.200000
d[3]=4.200000
d[4]=5.200000
d[5]=6.200000
b=abcdef

rect1:
  pt1: -4, -5
  pt2: 77, 21
rect2:
  pt1: -344, -732
  pt2: 233, 99

Depth: 32
Stop: 4
NodeType: -1
funct: 1
TypeData: -1
Index: -87
Nchildren: 45
Data: 7.450000
Parent: 0

fd: -356
ino: 99999999
name: AutoMap.yxx
```

For the CTp `Treenode`, AutoLink is needed to send the whole tree, just by giving the root.

A Nested Structure

Here are the type declarations:

```
/*~ AutoMap_Begin */

struct day_of_week
{
  char    day_name[10];
  int     day_date;
```

```
};

typedef struct day_of_week dow;


struct time_day
{
  int     hour;
  int     minute;
  int     second;
};

typedef struct time_day  td;


struct weather
{
  char    kind[10];
  int    temperature;
};

typedef struct weather  wt;


struct place
{
  char    town[15];
  char    state[15];
  char    country[20];
};

typedef struct place   pl;


struct Day
{
  dow   _day;
  td    _time;
  wt    _weather;
  pl    _place;
};

typedef struct Day   day /*~ AutoMap_CTpUsed */;


/*~ AutoMap_End */
```

This produces output file mpitypes.c:

```
#define MaxFields       50

MPI_Datatype    AutoMap_day;

void Build_MPI_Types()
{
day     CType0;
```

```
MPI_Aint        disp[MaxFields];
MPI_Datatype    type[MaxFields];
int             blocklen[MaxFields];
int             base;
int             i;

dow     TpDef0;
MPI_Datatype    AutoMap_dow;
td      TpDef1;
MPI_Datatype    AutoMap_td;
wt      TpDef2;
MPI_Datatype    AutoMap_wt;
pl      TpDef3;
MPI_Datatype    AutoMap_pl;

/* dow */
MPI_Address(TpDef0.day_name,    &disp[0]);
MPI_Address(&TpDef0.day_date,   &disp[1]);
type[0] = MPI_CHAR;
type[1] = MPI_INT;
blocklen[0] = 1*(10);
blocklen[1] = 1;
base = disp[0]; for(i=0; i<2; i++) disp[i]-=base;
MPI_Type_struct(2, blocklen, disp, type, &AutoMap_dow);
MPI_Type_commit(&AutoMap_dow);

/* td */
MPI_Address(&TpDef1.hour,       &disp[0]);
MPI_Address(&TpDef1.minute,     &disp[1]);
MPI_Address(&TpDef1.second,     &disp[2]);
type[0] = MPI_INT;
type[1] = MPI_INT;
type[2] = MPI_INT;
blocklen[0] = 1;
blocklen[1] = 1;
blocklen[2] = 1;
base = disp[0]; for(i=0; i<3; i++) disp[i]-=base;
MPI_Type_struct(3, blocklen, disp, type, &AutoMap_td);
MPI_Type_commit(&AutoMap_td);

/* wt */
MPI_Address(TpDef2.kind,        &disp[0]);
MPI_Address(&TpDef2.temperature,        &disp[1]);
type[0] = MPI_CHAR;
type[1] = MPI_INT;
blocklen[0] = 1*(10);
blocklen[1] = 1;
base = disp[0]; for(i=0; i<2; i++) disp[i]-=base;
MPI_Type_struct(2, blocklen, disp, type, &AutoMap_wt);
MPI_Type_commit(&AutoMap_wt);

/* pl */
MPI_Address(TpDef3.town,        &disp[0]);
MPI_Address(TpDef3.state,       &disp[1]);
MPI_Address(TpDef3.country,     &disp[2]);
type[0] = MPI_CHAR;
```

```
type[1] = MPI_CHAR;
type[2] = MPI_CHAR;
blocklen[0] = 1*(15);
blocklen[1] = 1*(15);
blocklen[2] = 1*(20);
base = disp[0]; for(i=0; i<3; i++) disp[i]-=base;
MPI_Type_struct(3, blocklen, disp, type, &AutoMap_pl);
MPI_Type_commit(&AutoMap_pl);

/* day */
MPI_Address(&CType0._day,       &disp[0]);
MPI_Address(&CType0._time,      &disp[1]);
MPI_Address(&CType0._weather,   &disp[2]);
MPI_Address(&CType0._place,     &disp[3]);
type[0] = AutoMap_dow;
type[1] = AutoMap_td;
type[2] = AutoMap_wt;
type[3] = AutoMap_pl;
blocklen[0] = 1;
blocklen[1] = 1;
blocklen[2] = 1;
blocklen[3] = 1;
base = disp[0]; for(i=0; i<4; i++) disp[i]-=base;
MPI_Type_struct(4, blocklen, disp, type, &AutoMap_day);
MPI_Type_commit(&AutoMap_day);
}
```

3.2.2 With AutoLink

id3

This program uses a Tree, as a linked structure. To send it over MPI, with AutoLink, add marks in the code for AutoMap to extract correctly the information needed.

```
/*~ AutoMap_Begin */

#define SIZE 14
#define ATTRIBUTES 5

typedef struct tnode *Treeptr;

typedef struct tnode {
  unsigned char mask[SIZE];
  unsigned char Ataken[ATTRIBUTES];
  int Depth;
  int Stop;
  int PropertyType;
  double InfoRem;
  double InfoParent;
  unsigned char InfoRemCalc;
  int PropertyOfParent;
  int PropertyValueOfParent;
```

```
  int Nchildren;
  Treeptr child;
  Treeptr sibling;
} Treenode /*~ AutoMap_CTpUsed */;

/*~ AutoMap_End */
```

This is the output on the screen from running AutoMap:

```
-- AutoMap for C files --
First parse completed successfully.
AST constructed
Main parse completed successfully.
num_typedefs:   2
num_structures: 1




-----------------------------------------------------------------
--------   This is the AutoLink Configuration program   ---------
-----------------------------------------------------------------




---   Traversing Graphs  ---

Do you intend to use the AutoLink-traversing routine? (Y/N) ->n




---   Memory Management   ---

Do you wish that AutoLink uses your own memory management routines? (Y/N)->y

Please enter the names of the instance allocating routines
for each datatype, or ENTER for none:
            Treenode: getnode

Configuration of AutoLink complete.


0 Errors, 0 Warnings, Highest Severity 0
```

After you answer the questions, the program ends and output three files:
userdefs.h:

```
#ifndef USERDEF
```

```
#define USERDEF

/* TYPES */


#define SIZE 14
#define ATTRIBUTES 5

typedef struct tnode *Treeptr;

typedef struct tnode {
  unsigned char mask[SIZE];
  unsigned char Ataken[ATTRIBUTES];
  int Depth;
  int Stop;
  int PropertyType;
  double InfoRem;
  double InfoParent;
  unsigned char InfoRemCalc;
  int PropertyOfParent;
  int PropertyValueOfParent;
  int Nchildren;
  Treeptr child;
  Treeptr sibling;
} Treenode ;



/* DEFINES */

#define MaxFields        50
#define CTpUsed          1
#define STpUsed          0
#define HTpUsed          0

typedef Treenode         CTp0;   /* datatype number 0 */

#define CT0

/* Treenode */
#define PtrFld00         child
#define PtrTp00          0
#define PtrFld01         sibling
#define PtrTp01          0
#define GetCTp0          getnode

#define BSize0           100

int BSize[CTpUsed+STpUsed+HTpUsed+1] = { BSize0, 0 };

int BSizeb[STpUsed+HTpUsed+1] = { 0 };

#endif
```

The section `/* TYPES */` is a copy of the input file. The section `/* DEFINES */` is the output of AutoMap, to be used by AutoLink. Note that the "`+1`" in the size field of the arrays `BSize` and `BSizeb` is here to prevent the declaration of empty arrays. AutoLink takes care of the fields `child` and `sibling`, which are links to other nodes of the tree. They are declared as pointer fields to instances of type #0, which is `Treenode`.
mpitypes.c:

```
void Build_MPI_Types()
{
CTp0 CType0;

MPI_Aint        disp[MaxFields];
MPI_Datatype    type[MaxFields];
int             blocklen[MaxFields];
int             base;
int             i;


/* CTp0 */
MPI_Address(CType0.mask,        &disp[0]);
MPI_Address(CType0.Ataken,      &disp[1]);
MPI_Address(&CType0.Depth,      &disp[2]);
MPI_Address(&CType0.Stop,       &disp[3]);
MPI_Address(&CType0.PropertyType,       &disp[4]);
MPI_Address(&CType0.InfoRem,    &disp[5]);
MPI_Address(&CType0.InfoParent, &disp[6]);
MPI_Address(&CType0.InfoRemCalc,        &disp[7]);
MPI_Address(&CType0.PropertyOfParent,   &disp[8]);
MPI_Address(&CType0.PropertyValueOfParent,      &disp[9]);
MPI_Address(&CType0.Nchildren,  &disp[10]);
MPI_Address(&CType0.child,      &disp[11]);
MPI_Address(&CType0.sibling,    &disp[12]);
type[0] = MPI_UNSIGNED_CHAR;
type[1] = MPI_UNSIGNED_CHAR;
type[2] = MPI_INT;
type[3] = MPI_INT;
type[4] = MPI_INT;
type[5] = MPI_DOUBLE;
type[6] = MPI_DOUBLE;
type[7] = MPI_UNSIGNED_CHAR;
type[8] = MPI_INT;
type[9] = MPI_INT;
type[10] = MPI_INT;
type[11] = MPI_INT;
type[12] = MPI_INT;
blocklen[0] = 1*(SIZE);
blocklen[1] = 1*(ATTRIBUTES);
blocklen[2] = 1;
blocklen[3] = 1;
blocklen[4] = 1;
blocklen[5] = 1;
blocklen[6] = 1;
blocklen[7] = 1;
```

```
blocklen[8] = 1;
blocklen[9] = 1;
blocklen[10] = 1;
blocklen[11] = 1;
blocklen[12] = 1;
base = disp[0]; for(i=0; i<13; i++) disp[i]-=base;
MPI_Type_struct(13, blocklen, disp, type, &MPI_Types[0]);
MPI_Type_commit(&MPI_Types[0]);
}
```

This file contains the creation of the MPI Datatype that is going to be used by AutoLink to send the tree.
automap.h:

```
/* AutoMap DEFINES */

#define Treenode_AutoNbr        0
```

This is used to reference the types created in `mpitypes.c` with the array `MPI_Types`. Include these files in your code and compile both your code and AutoLink before linking them together. This can be done with a suitable makefile. For more details on this test, and the results of AutoLink, see [6]. This same example has been run on the last version of AutoMap and the output files are still the same, except that the section `/* TYPES */` in `userdefs.h` is a copy of the code between the AutoMap marks, without the comments, so the user doesn't have to put the declarations in a separate file anymore.

3.3 Future Work

There are still some restrictions on the AutoMap grammar. For instance, the grammar for constant expressions should be improved (the use of '+' should be allowed). The use of nested structures needs to be more flexible (not necessarily with typedef names). The MPI type for an enumeration should be `MPI_INT`. The AutoLink configuration questions should include something to customize the `BSize`, and also to handle the unions.

AutoMap reads only one file: if one wants to include header files in the code and use the types declared in these files, AutoMap has to read them. However, AutoMap can't be run several times at present. The easiest solution is to run a preprocessor on the file to actually include the header files in the code. This preprocessor must perform the inclusion, but it must keep the comments where the AutoMap directives reside. Such a preprocessor can be implemented by another grammar that would come before AutoStart.

Finally, AutoMap needs to be able to handle *unions* so that they can be sent across heterogeneous nodes.

Appendix A

AutoMap Summary

```
Summary of AutoMap requirements and outputs
-------------------------------------------

User responsibility:

   Place around structs to be converted to MPI data types

            /*~ AUTOMAP_BEGIN */

            /*~ AUTOMAP_END */


   Place next to struct definitions and BEFORE the struct semicolon
            /*~ AUTOMAP_CTpUsed   */           ;
            /*~ AUTOMAP_ATpUsed   */           ;
            /*~ AUTOMAP_HTpUsed   */           ;


Input files: user code, suitably marked


Output files: userdefs.h mpitypes.c autolink.h


AutoMap names MPI datatypes: AUTOMAP_ +  C struct name
If the C struct name is X, the MPI datatype is: AUTOMAP_X


Files for creation of AutoMap with yacc++:
   AutoStart.yxx AutoMap.yxx AutoMap_main.cxx
```

Appendix B

Makefile

```
###############################################################################
#
#    FILENAME:           makefile
#
#                        **  SGI IRIX VERSION  **
#
#    FILE DESCRIPTION:   makefile for AutoMap 1.1
#
#
#       input grammars:   AutoSart.yxx
#                         AutoMap.yxx
#
# main program:     AutoMap_main.cxx
#
###############################################################################

#
#  change "YY_MYDIR" to point to your current working directory
#

YY_MYDIR=.
YY_MYLIB=$(YY_MYDIR)/lib.sgi
YY_MYINC=$(YY_MYDIR)/inc.sgi

#
#  change "YXXDIR" or "YY_LIB" and "YY_INC" to point to the yxx sources
#

YXXDIR=/yxx
YY_LIB=$(YXXDIR)/lib.sgi
YY_INC=$(YXXDIR)/inc.sgi
YY_LIBRARY=$(YY_LIB)/yy_lol.a

#
#  change "YXX", "YY_ETBL", and "YY_STBL" to point to the yxx executables
#
```

```
YXX=yxx
YY_ETBL=yy_etbl
YY_STBL=yy_stbl

#
#  Compilation options.
#

YXXFLAGS= -debug -t r
DEFINES=
INCLUDES= -I$(YY_MYDIR) -I$(YY_MYINC) -I$(YY_INC)
CXXFLAGS=
CXX=CC
MV=mv
RM=rm

#
#  Rules for building the primary output files
#

all: AutoMap

AutoMap: AutoStart.yxx AutoMap.yxx AutoMap_main.cxx $(YY_LIBRARY)
$(YXX) AutoStart.yxx $(YXXFLAGS)
$(YXX) AutoMap.yxx $(YXXFLAGS)
-$(CXX) $(CXXFLAGS) $(DEFINES) $(INCLUDES) -c AutoMap_main.cxx -o yy_main.o
-$(CXX) $(CXXFLAGS) $(DEFINES) $(INCLUDES) -c AM1_lex.cxx -o AM1_lex.o
-$(CXX) $(CXXFLAGS) $(DEFINES) $(INCLUDES) -c AM1_psr.cxx -o AM1_psr.o
-$(CXX) $(CXXFLAGS) $(DEFINES) $(INCLUDES) -c AM2_lex.cxx -o AM2_lex.o
-$(CXX) $(CXXFLAGS) $(DEFINES) $(INCLUDES) -c AM2_psr.cxx -o AM2_psr.o
-$(CXX) $(CXXFLAGS) -o AutoMap yy_main.o AM1_lex.o AM1_psr.o AM2_lex.o AM2_psr.o $(YY_LIBRARY)
-$(RM) AM1_lex.o
-$(RM) AM1_lex.cxx
-$(RM) AM1_lex.h
-$(RM) AM1_psr.o
-$(RM) AM1_psr.cxx
-$(RM) AM1_psr.h
-$(RM) AM2_lex.o
-$(RM) AM2_lex.cxx
-$(RM) AM2_lex.h
-$(RM) AM2_psr.o
-$(RM) AM2_psr.cxx
-$(RM) AM2_psr.h
-$(RM) yy_myref.h
-$(RM) yy_main.o

# end of makefile
```

Appendix C

Makefile for stand-alone version

```
##############################################################################
#
#    FILENAME:           makefile
#
#                        **  SGI IRIX VERSION  **
#
#    FILE DESCRIPTION:   makefile for AutoMap 1.1 stand-alone
#
#
#       input grammars:   AutoSart.yxx
#                         AutoMap_sta.yxx
#
# main program:     AutoMap_main.cxx
#
##############################################################################

#
#  change "YY_MYDIR" to point to your current working directory
#

YY_MYDIR=.
YY_MYLIB=$(YY_MYDIR)/lib.sgi
YY_MYINC=$(YY_MYDIR)/inc.sgi

#
#  change "YXXDIR" or "YY_LIB" and "YY_INC" to point to the yxx sources
#

YXXDIR=/yxx
YY_LIB=$(YXXDIR)/lib.sgi
YY_INC=$(YXXDIR)/inc.sgi
YY_LIBRARY=$(YY_LIB)/yy_lol.a

#
#  change "YXX", "YY_ETBL", and "YY_STBL" to point to the yxx executables
#
```

```
YXX=yxx
YY_ETBL=yy_etbl
YY_STBL=yy_stbl

#
#  Compilation options.
#

YXXFLAGS= -debug -t r
DEFINES=
INCLUDES= -I$(YY_MYDIR) -I$(YY_MYINC) -I$(YY_INC)
CXXFLAGS=
CXX=CC
MV=mv
RM=rm

#
#  Rules for building the primary output files
#

all: AutoMap

AutoMap: AutoStart.yxx AutoMap_sta.yxx AutoMap_main.cxx $(YY_LIBRARY)
$(YXX) AutoStart.yxx $(YXXFLAGS)
$(YXX) AutoMap_sta.yxx $(YXXFLAGS)
-$(CXX) $(CXXFLAGS) $(DEFINES) $(INCLUDES) -c AutoMap_main.cxx -o yy_main.o
-$(CXX) $(CXXFLAGS) $(DEFINES) $(INCLUDES) -c AM1_lex.cxx -o AM1_lex.o
-$(CXX) $(CXXFLAGS) $(DEFINES) $(INCLUDES) -c AM1_psr.cxx -o AM1_psr.o
-$(CXX) $(CXXFLAGS) $(DEFINES) $(INCLUDES) -c AM2_lex.cxx -o AM2_lex.o
-$(CXX) $(CXXFLAGS) $(DEFINES) $(INCLUDES) -c AM2_psr.cxx -o AM2_psr.o
-$(CXX) $(CXXFLAGS) -o AutoMap yy_main.o AM1_lex.o AM1_psr.o AM2_lex.o AM2_psr.o $(YY_LIBRARY)
-$(RM) AM1_lex.o
-$(RM) AM1_lex.cxx
-$(RM) AM1_lex.h
-$(RM) AM1_psr.o
-$(RM) AM1_psr.cxx
-$(RM) AM1_psr.h
-$(RM) AM2_lex.o
-$(RM) AM2_lex.cxx
-$(RM) AM2_lex.h
-$(RM) AM2_psr.o
-$(RM) AM2_psr.cxx
-$(RM) AM2_psr.h
-$(RM) yy_myref.h
-$(RM) yy_main.o

# end of makefile
```

Appendix D

AutoStart.yxx

```
////////////////////////////////////////////////////////////////////////////////
//
//  FILENAME:           AutoStart.yxx
//
//  FILE DESCRIPTION:   Grammar to detect the Begin mark for AutoMap
//
//
//  version 1.1
//
//  Eric Baland         January 1997
////////////////////////////////////////////////////////////////////////////////


OUTPUT LEXER HEADER "AM1_lex.h";
OUTPUT LEXER SOURCE "AM1_lex.cxx";
OUTPUT PARSER HEADER "AM1_psr.h";
OUTPUT PARSER SOURCE "AM1_psr.cxx";



class AutoStart;

lexer :: NAME AM1_lexer; // lexer declarations and definitions

local {
        static int to_quit=0;
}

                // token list
token           IDENTIFIER
                ;

                // Tokens to be ignored
discard token   COMMENT  ANYCHAR
                ;


// Token definitions
```

```
IDENTIFIER   :  (("a" .. "z" | "A" .. "Z" | "_")
                ("a" .. "z" | "A" .. "Z" | "0" .. "9" | "_")*)
             ;

COMMENT      :   "/*" ( "~" (@)* AutoMap_stmt ( (@)* "*" )+ "/"
                      |  ( (@)* "*" )+ "/" )
             ;
                { if (to_quit) yy_lex_quit(); }

AutoMap_stmt :   "AutoMap_"
                ( "Begin" { to_quit=1; }
                | @ )
             ;

ANYCHAR      :   @ | "/"
             ;


parser  :: NAME AM1_parser    // parser definitions
;

////////////////////////////////////////////////////////////////////////////////
// Beginning of the parser rules                                              //
////////////////////////////////////////////////////////////////////////////////

start   : (IDENTIFIER)*
        ;
```

Appendix E

AutoMap.yxx

```
///////////////////////////////////////////////////////////////////////////////
//
//  FILENAME:           AutoMap.yxx
//
//  FILE DESCRIPTION:   Input grammar for parsing C declarations
//                      Built the corresponding Abstract Syntax Tree (AST)
//
//  version 1.1
//
//  Eric Baland         january 1997
///////////////////////////////////////////////////////////////////////////////
//
//  Please, note the way the parser is stopped from the lexer
//  in the token COMMENT
//
///////////////////////////////////////////////////////////////////////////////


OUTPUT LEXER HEADER "AM2_lex.h";
OUTPUT LEXER SOURCE "AM2_lex.cxx";
OUTPUT PARSER HEADER "AM2_psr.h";
OUTPUT PARSER SOURCE "AM2_psr.cxx";

class AutoMap;

lexer :: NAME AM2_lexer; // lexer declarations and definitions


                // list of keywords
keyword         struct char int float void short long double signed unsigned
                const volatile typedef union enum auto register static extern
                ;

                // Token list
token           LBRACE  IDENTIFIER  INTEGER_CST  RBRACE  SEMI  COLON  LBRACK
                RBRACK  STAR  COMMA  EQUAL  LPAR  RPAR ;

                // Tokens to be ignored
```

```
discard token   WHITE_SPACE  COMMENT  define_stmt  include_stmt
                ;


// construct declarations
construct    IDENTIFIER :: keyword

  base { public struct_base_class }

  constructor body {
    yy_symbol()->yy_sym_type(yy_symbol()->yy_sym_tkn_type());
  } ;

// Token definitions

IDENTIFIER   :  (("a" .. "z" | "A" .. "Z" | "_")
                ("a" .. "z" | "A" .. "Z" | "0" .. "9" | "_")*)
                      { ident_ptr = new char[yy_lex_len() + 1];
                        memcpy(ident_ptr, yy_lex_token(), yy_lex_len());
                        ident_ptr[yy_lex_len()] = '\0';
                        if (read_struct == 1) { read_struct=0;
                          struct_name = new char[yy_lex_len()+1];
                          memcpy(struct_name, yy_lex_token(), yy_lex_len());
                          struct_name[yy_lex_len()] = '\0'; }
                        fprintf(header_ptr, ident_ptr); }
             ;

INTEGER_CST  : (("1" .. "9")  ("0" .. "9")*
                |"0"  ("0" .. "7")*
                | ("0x" | "0X") ("0" .. "9" | "a" .. "f" | "A" .. "F")+ )
               (("u"  |  "U")?  ("l"  |  "L")?
                |("l"  |  "L")  ("u"  |  "U"))
             ;
                        { ident_ptr = new char[yy_lex_len() + 1];
                          memcpy(ident_ptr, yy_lex_token(), yy_lex_len());
                          ident_ptr[yy_lex_len()] = '\0';
                          fprintf(header_ptr, ident_ptr); }

LBRACE       :   "{"
             ;
        { fputc(*yy_lex_token(), header_ptr); }

RBRACE       :   "}"
             ;
        { fputc(*yy_lex_token(), header_ptr); }

SEMI         :   ";"
             ;
        { fputc(*yy_lex_token(), header_ptr); }

COLON        :   ":"
             ;
        { fputc(*yy_lex_token(), header_ptr); }

COMMA        :   ","
             ;
        { fputc(*yy_lex_token(), header_ptr); }
```

```
STAR         :   "*"
             ;
        { fputc(*yy_lex_token(), header_ptr); }

LBRACK       :   "["
             ;
        { fputc(*yy_lex_token(), header_ptr); }

RBRACK       :   "]"
             ;
        { fputc(*yy_lex_token(), header_ptr); }

EQUAL        :   "="
             ;
        { fputc(*yy_lex_token(), header_ptr); }

LPAR         :   "("
             ;
        { fputc(*yy_lex_token(), header_ptr); }

RPAR         :   ")"
             ;
        { fputc(*yy_lex_token(), header_ptr); }

WHITE_SPACE  :   (" "             // space
             |    "\t"            // tab
             |    "\n"            // newline
                                  {  ++yy_cur_lineno();  }
                  )+
             ;
        { token_ptr = new char[yy_lex_len() + 1];
          memcpy(token_ptr, yy_lex_token(), yy_lex_len());
          token_ptr[yy_lex_len()] = '\0';
          fprintf(header_ptr, token_ptr); }

COMMENT      : "/*" ( "~" (@)* AutoMap_stmt ( (@)* "*" )+ "/"
                      |    ( (@)* "*" )+ "/" )
             ;
               { if (to_quit_psr) {
                    yy_this_psr_obj->yy_psr_quit();
                    yy_lex_rdc() = yy_eof_; }
                }


AutoMap_stmt :   "AutoMap_" ( "CTpUsed"  { ctpused=1; }
                            | "ATpUsed"  { atpused=1; }
                            | "HTpUsed"  { htpused=1; }
                            | "End"      { to_quit_psr=1; }
                            | @ )
             ;

define_stmt  :   "#define" @* "\n" { ++yy_cur_lineno(); }
             ;
        { token_ptr = new char[yy_lex_len() + 1];
          memcpy(token_ptr, yy_lex_token(), yy_lex_len());
          token_ptr[yy_lex_len()] = '\0';
```

```
          fprintf(header_ptr, token_ptr); }

include_stmt :   "#include" @* "\n" { ++yy_cur_lineno(); }
             ;
        { token_ptr = new char[yy_lex_len() + 1];
          memcpy(token_ptr, yy_lex_token(), yy_lex_len());
          token_ptr[yy_lex_len()] = '\0';
          fprintf(header_ptr, token_ptr); }

parser  :: NAME AM2_parser // parser definitions
    destructor {
        int i;

       // delete the objects in the parser stack
       // each object recursively deletes its children

        for (i = 1; i < yy_psr_last(); ++i) {
          if (yy_psr_ref(i).as_base_ptr != NULL) {
            //delete yy_psr_ref(i).as_base_ptr;
          }
        }
    } ;

global {
#include <iostream.h>
#include <stdio.h>
#include <string.h>
#include "yy_sym.h"

#define STRUCT_TYPE             1
#define UNION_TYPE              2
#define TYPEDEF_STRUCT_TYPE     3
#define TYPEDEF_STRUCT_TYPE2    4
#define TYPEDEF_UNION_TYPE      5
#define TYPEDEF_UNION_TYPE2     6
#define TYPEDEF_TYPE            7

////////////////////////////////////////////////////////////////////////////////
// NON-STAND-ALONE Version
////////////////////////////////////////////////////////////////////////////////
//#define STANDALONE

typedef class struct_base_class *base_ptr;
typedef class struct_decl_file  *decl_file_ptr;
typedef class declaration       *declaration_ptr;
typedef class TYPE_SPECIF       *type_specif_ptr;
typedef class TYPEDEF_NAME      *typedef_name_ptr;
typedef class INIT_DECLARATOR   *typedef_ptr;
typedef class struct_or_union_specif  *struct_ptr;
typedef class STRUCT_DECL       *decl_ptr;
typedef class STRUCT_DECLARATOR *declarator_ptr;

#ifndef yy_psr_code_
extern
#endif
int     num_var,        // number of variables (fields) in a structure
        num_declarator, // number of struct_declarators in a STRUCT_DECL
```

```
        num_star,       // number of stars in a declarator
        num_struct,     // number of structures
        num_union,      // number of unions
        num_typedef,    // number of typedefs not on structures or unions
        num_typedef_struct, // number of typedefs on structures
        num_typedef_union,  // number of typedefs on unions
        num_decl,       // number of STRUCT_DECLs in a structure or union
        ctpused,
        atpused,
        htpused,
        array,
        class_type,
        to_quit_psr,
        read_struct;
#ifndef yy_psr_code_
extern
#endif
char    *name_ptr, *ident_ptr, *size_ptr, *struct_name
#ifndef STANDALONE
, *token_ptr
#endif
;

#ifndef STANDALONE
#ifndef yy_psr_code_
extern
#endif
FILE *header_ptr;
#endif

}

union   {
          base_ptr              as_base_ptr;
          decl_file_ptr         as_decl_file_ptr;
          declaration_ptr       as_declaration_ptr;
          type_specif_ptr       as_specif_ptr;
          typedef_name_ptr      as_typedef_name_ptr;
          typedef_ptr           as_typedef_ptr;
          struct_ptr            as_struct_ptr;
          decl_ptr              as_decl_ptr;
          declarator_ptr        as_declarator_ptr;
        }


//construct declaration for base class

base construct struct_base_class ::

  member {
    public:
        virtual yy_sym_ptr yy_symbol()          { return(NULL); }
        virtual const char *display_name()      { return(NULL); }
        virtual int        type()               { return(0); }
  };
```

```
//other construct declarations

construct struct_decl_file :: all

base { public struct_base_class }

constructor body {
        cout << "AST constructed" << endl;
}

constructor initializer {
  num_structures (num_struct),
  num_typedefs  (num_typedef),
  num_unions    (num_union),
  num_typedef_structs   (num_typedef_struct),
  num_typedef_unions    (num_typedef_union) }

member {
  public:
    int         num_structures;
    int         num_typedefs;
    int         num_unions;
    int         num_typedef_structs;
    int         num_typedef_unions;
};


construct declaration :: all

base { public struct_base_class }

constructor initializer {
  decl_type (class_type) }

constructor body {
  switch (class_type) {
    case STRUCT_TYPE          : ++num_struct; break;
    case UNION_TYPE           : ++num_union; break;
    case TYPEDEF_STRUCT_TYPE  : ++num_typedef_struct; break;
    case TYPEDEF_STRUCT_TYPE2 : ++num_typedef_struct; ++num_struct; break;
    case TYPEDEF_UNION_TYPE   : ++num_typedef_union; break;
    case TYPEDEF_UNION_TYPE2  : ++num_typedef_union; ++num_union; break;
    case TYPEDEF_TYPE         : ++num_typedef; break;
  }
}

member {
  public:
    int         decl_type;
    int         type() { return(decl_type); }
};

construct STORAGE_SPECIF :: default

base { public struct_base_class }

constructor initializer {
```

```
  name (ident_ptr) }

member {
  public:
    char        *name;
    const char  *display_name() { return(name); }
};

construct TYPE_SPECIF :: all

base { public struct_base_class }

constructor initializer {
  name (ident_ptr) }

member {
  public:
    char        *name;
    const char  *display_name() { return(name); }
};

construct TYPE_QUALIF :: default

base { public struct_base_class }

constructor initializer {
  name (ident_ptr) }

member {
  public:
    char        *name;
    const char  *display_name() { return(name); }
};

construct INIT_DECLARATOR :: default

base { public struct_base_class }

constructor initializer {
  num_stars     (num_star),
  is_array      (array),
  array_size    (size_ptr),
  name          (name_ptr),
  MPI_Construct         (0),
  AutoMap_CTpUsed       (ctpused),
  AutoMap_ATpUsed       (atpused),
  AutoMap_HTpUsed       (htpused) }

member {
  public:
    int         num_stars;
    int         is_array;
    int         MPI_Construct; // 1 if the MPI_Datatype already declared
                               // 2 if the MPI_Datatype already defined
    int         AutoMap_CTpUsed;        // 1 if used as CTp
    int         AutoMap_ATpUsed;        // 1 if used as ATp
    int         AutoMap_HTpUsed;        // 1 if used as HTp
```

```
    char        *array_size;
    char        *name;
    const char  *display_name() { return(name); }
};

construct TYPEDEF_NAME :: default

base {public struct_base_class }

constructor initializer {
  name (name_ptr) }

member {
  public:
    char        *name;
    const char  *display_name() { return(name); }
};

construct struct_or_union_specif :: all

base { public struct_base_class }

constructor initializer {
  name (struct_name),
  num_fields (num_var),
  num_pointers (0),
  num_decls (num_decl) }

member {
  public:
    char        *name;
    int         num_fields;
    int         num_pointers;
    int         num_decls;
    const char  *display_name() { return(name); }
};

construct enum_specifier :: default

base { public struct_base_class }
;

construct STRUCT_DECL :: all

base { public struct_base_class }

constructor initializer {
  num_declarators (num_declarator) }

constructor body {
  ++num_decl;
}

member {
  public:
    int         num_declarators;
};
```

```
construct STRUCT_DECLARATOR :: default

base { public struct_base_class }

constructor initializer {
  num_stars (num_star),
  AutoMap_Ptr (0),  // 1 if you want to send what the field points to
  is_array (array),
  array_size (size_ptr),
  name (name_ptr) }

member {
  public:
    int         num_stars;
    int         AutoMap_Ptr;
    int         is_array;
    char        *array_size;
    char        *name;
    const char  *display_name() { return(name); }
};


///////////////////////////////////////////////////////////////////////////////
// Beginning of the parser rules                                            //
///////////////////////////////////////////////////////////////////////////////

start struct_decl_file;

struct_decl_file  : { num_struct=0; num_union=0; num_typedef=0;
                      num_typedef_struct=0; num_typedef_union=0;
                      to_quit_psr=0; }
                   ( declaration )*
                  ;

declaration  :  { class_type=0; ctpused=0; atpused=0; htpused=0;
                  name_ptr=NULL; read_struct=0; }
                ( STORAGE_SPECIF | TYPE_QUALIF )*
                ( TYPEDEF_NAME | TYPE_SPECIF )
                ( STORAGE_SPECIF | TYPE_QUALIF | TYPE_SPECIF )*
                ( INIT_DECLARATOR  (COMMA  INIT_DECLARATOR)* )?
                SEMI
             ;

STORAGE_SPECIF :  auto | register | static | extern
               |  typedef  { class_type = TYPEDEF_TYPE; }
               ;

TYPE_SPECIF  :  char | int | float | void | short | long | double | signed
             |  unsigned
             |  struct_or_union_specif
             |  enum_specifier
             ;

TYPE_QUALIF  :  const | volatile
             ;
```

```
INIT_DECLARATOR :  DECLARATOR ( EQUAL INITIALIZER )?
                ;

struct_or_union_specif :
                struct_or_union { struct_name=NULL; read_struct=1; }
              ( IDENTIFIER ?
                     LBRACE  { num_var=0; num_decl=0; read_struct=0; }
                    ( STRUCT_DECL )+
                     RBRACE
        { if (class_type == TYPEDEF_STRUCT_TYPE)
            class_type=TYPEDEF_STRUCT_TYPE2;
          if (class_type == TYPEDEF_UNION_TYPE)
            class_type=TYPEDEF_UNION_TYPE2; }
              | IDENTIFIER
        { if (class_type == STRUCT_TYPE || class_type == UNION_TYPE)
            class_type=0; } )
             ;

struct_or_union : struct
        { if (class_type == TYPEDEF_TYPE) class_type=TYPEDEF_STRUCT_TYPE;
          else if (class_type == 0) class_type=STRUCT_TYPE; }
                | union
        { if (class_type == TYPEDEF_TYPE) class_type=TYPEDEF_UNION_TYPE;
          else if (class_type == 0) class_type=UNION_TYPE; }
                ;

STRUCT_DECL  :  { num_declarator=0; }
                ( TYPE_QUALIF )*
                ( TYPEDEF_NAME | TYPE_SPECIF )
                ( TYPE_SPECIF | TYPE_QUALIF )*
                STRUCT_DECLARATOR  ( COMMA  STRUCT_DECLARATOR )*
                SEMI
             ;

STRUCT_DECLARATOR : DECLARATOR
                  | DECLARATOR ?  COLON  CST_EXPR
                  ;

DECLARATOR   :  { num_star=0; array=0; size_ptr = new char[2];
                  strcpy(size_ptr, "1"); }
                ( POINTER ) ?
                  DIRECT_DECL
             ;

DIRECT_DECL  :  IDENTIFIER
                        { ++num_var; ++num_declarator;
                          name_ptr=ident_ptr; }
             |  LPAR  DECLARATOR  RPAR
             |  DIRECT_DECL
                ( LBRACK  ( CST_EXPR  { strcat(size_ptr, "*");
                                        strcat(size_ptr, "(");
                                        strcat(size_ptr, ident_ptr);
                                        strcat(size_ptr, ")"); } )?  RBRACK
                  { array=1; }
                | LPAR  ( IDENTIFIER ( COMMA  IDENTIFIER )* ) ?  RPAR
                | LPAR  PARAMETER_DECL  ( COMMA  PARAMETER_DECL )*  RPAR )
             ;
```

```
POINTER       :  STAR { ++num_star; } ( TYPE_QUALIF | STAR { ++num_star; } )*
              ;

PARAMETER_DECL : ( STORAGE_SPECIF | TYPE_QUALIF )*
                 ( TYPEDEF_NAME | TYPE_SPECIF )
                 ( STORAGE_SPECIF | TYPE_QUALIF | TYPE_SPECIF )*
                 ( DECLARATOR | ABSTRACT_DECLARATOR ) ?
               ;

ABSTRACT_DECLARATOR : POINTER
                    | POINTER ?  DIRECT_ABST_DECL
                    ;

DIRECT_ABST_DECL    :  LPAR  ABSTRACT_DECLARATOR  RPAR
                    |  DIRECT_ABST_DECL ?
                       ( LBRACK  CST_EXPR ?  RBRACK
                       | LPAR  (PARAMETER_DECL (COMMA PARAMETER_DECL)* )? RPAR)
                    ;

TYPEDEF_NAME :  IDENTIFIER { name_ptr=ident_ptr; }
              ;

enum_specifier :  enum
                  IDENTIFIER ?
                    LBRACE
                    ENUMERATOR  ( COMMA  ENUMERATOR )*  RBRACE
               |  enum
                  IDENTIFIER
               ;

ENUMERATOR    :  IDENTIFIER  ( EQUAL CST_EXPR )?
              ;

CST_EXPR      :  INTEGER_CST
              |  IDENTIFIER
              ;

INITIALIZER   :  CST_EXPR
              |  LBRACE  INITIALIZER ( COMMA  INITIALIZER )*  COMMA ? RBRACE
              ;
```

Appendix F

AutoMap_sta.yxx

```
///////////////////////////////////////////////////////////////////////////////
//
//  FILENAME:           AutoMap_sta.yxx
//
//  FILE DESCRIPTION:   Input grammar for parsing C declarations
//                      Built the corresponding Abstract Syntax Tree (AST)
//
//  version 1.1
//  stand-alone
//
//  Eric Baland         january 1997
///////////////////////////////////////////////////////////////////////////////
//
//  Please, note the way the parser is stopped from the lexer
//  in the token COMMENT
//
///////////////////////////////////////////////////////////////////////////////


OUTPUT LEXER HEADER "AM2_lex.h";
OUTPUT LEXER SOURCE "AM2_lex.cxx";
OUTPUT PARSER HEADER "AM2_psr.h";
OUTPUT PARSER SOURCE "AM2_psr.cxx";

class AutoMap;

lexer :: NAME AM2_lexer; // lexer declarations and definitions


                // list of keywords
keyword         struct char int float void short long double signed unsigned
                const volatile typedef union enum auto register static extern
                ;

                // Token list
token           LBRACE  IDENTIFIER  INTEGER_CST  RBRACE  SEMI  COLON  LBRACK
                RBRACK  STAR  COMMA  EQUAL  LPAR  RPAR ;
```

```
                // Tokens to be ignored
discard token   WHITE_SPACE  COMMENT  define_stmt  include_stmt
                ;


// construct declarations
construct    IDENTIFIER :: keyword

  base { public struct_base_class }

  constructor body {
    yy_symbol()->yy_sym_type(yy_symbol()->yy_sym_tkn_type());
  } ;

// Token definitions

IDENTIFIER   :  (("a" .. "z" | "A" .. "Z" | "_")
                ("a" .. "z" | "A" .. "Z" | "0" .. "9" | "_")*)
                      { ident_ptr = new char[yy_lex_len() + 1];
                        memcpy(ident_ptr, yy_lex_token(), yy_lex_len());
                        ident_ptr[yy_lex_len()] = '\0';
                        if (read_struct == 1) { read_struct=0;
                          struct_name = new char[yy_lex_len()+1];
                          memcpy(struct_name, yy_lex_token(), yy_lex_len());
                          struct_name[yy_lex_len()] = '\0'; } }
             ;

INTEGER_CST  : (("1" .. "9")  ("0" .. "9")*
                |"0"  ("0" .. "7")*
                | ("0x" | "0X") ("0" .. "9" | "a" .. "f" | "A" .. "F")+ )
               (("u"  |  "U")?  ("l"  |  "L")?
                |("l"  |  "L")  ("u"  |  "U"))
             ;
                        { ident_ptr = new char[yy_lex_len() + 1];
                          memcpy(ident_ptr, yy_lex_token(), yy_lex_len());
                          ident_ptr[yy_lex_len()] = '\0'; }

LBRACE       :   "{"
             ;

RBRACE       :   "}"
             ;

SEMI         :   ";"
             ;

COLON        :   ":"
             ;

COMMA        :   ","
             ;

STAR         :   "*"
             ;

LBRACK       :   "["
             ;
```

```
RBRACK       :   "]"
             ;

EQUAL        :   "="
             ;

LPAR         :   "("
             ;

RPAR         :   ")"
             ;

WHITE_SPACE  :   (" "            // space
             |    "\t"           // tab
             |    "\n"           // newline
                                 {  ++yy_cur_lineno();  }
                  )+
             ;

COMMENT      : "/*" ( "~" (@)* AutoMap_stmt ( (@)* "*" )+ "/"
                      |   ( (@)* "*" )+ "/" )
             ;
               { if (to_quit_psr) {
                    yy_this_psr_obj->yy_psr_quit();
                    yy_lex_rdc() = yy_eof_; }
                }


AutoMap_stmt :   "AutoMap_" ( "CTpUsed"  { ctpused=1; }
                            | "ATpUsed"  { atpused=1; }
                            | "HTpUsed"  { htpused=1; }
                            | "End"      { to_quit_psr=1; }
                            | @ )
             ;

define_stmt  :   "#define" @* "\n" { ++yy_cur_lineno(); }
             ;

include_stmt :   "#include" @* "\n" { ++yy_cur_lineno(); }
             ;

parser  :: NAME AM2_parser // parser definitions
    destructor {
        int i;

       // delete the objects in the parser stack
       // each object recursively deletes its children

        for (i = 1; i < yy_psr_last(); ++i) {
          if (yy_psr_ref(i).as_base_ptr != NULL) {
            //delete yy_psr_ref(i).as_base_ptr;
          }
        }
    } ;

global {
```

```
#include <iostream.h>
#include <stdio.h>
#include <string.h>
#include "yy_sym.h"

///////////////////////////////////////////////////////////////////////////////
// STAND-ALONE Version
///////////////////////////////////////////////////////////////////////////////
#define STANDALONE

#define STRUCT_TYPE             1
#define UNION_TYPE              2
#define TYPEDEF_STRUCT_TYPE     3
#define TYPEDEF_STRUCT_TYPE2    4
#define TYPEDEF_UNION_TYPE      5
#define TYPEDEF_UNION_TYPE2     6
#define TYPEDEF_TYPE            7


typedef class struct_base_class *base_ptr;
typedef class struct_decl_file  *decl_file_ptr;
typedef class declaration       *declaration_ptr;
typedef class TYPE_SPECIF       *type_specif_ptr;
typedef class TYPEDEF_NAME      *typedef_name_ptr;
typedef class INIT_DECLARATOR   *typedef_ptr;
typedef class struct_or_union_specif  *struct_ptr;
typedef class STRUCT_DECL       *decl_ptr;
typedef class STRUCT_DECLARATOR *declarator_ptr;

#ifndef yy_psr_code_
extern
#endif
int     num_var,        // number of variables (fields) in a structure
        num_declarator, // number of struct_declarators in a STRUCT_DECL
        num_star,       // number of stars in a declarator
        num_struct,     // number of structures
        num_union,      // number of unions
        num_typedef,    // number of typedefs not on structures or unions
        num_typedef_struct, // number of typedefs on structures
        num_typedef_union,  // number of typedefs on unions
        num_decl,       // number of STRUCT_DECLs in a structure or union
        ctpused,
        atpused,
        htpused,
        array,
        class_type,
        to_quit_psr,
        read_struct;
#ifndef yy_psr_code_
extern
#endif
char    *name_ptr, *ident_ptr, *size_ptr, *struct_name
#ifndef STANDALONE
, *token_ptr
#endif
;
```

```
#ifndef STANDALONE
#ifndef yy_psr_code_
extern
#endif
FILE *header_ptr;
#endif

}

union   {
          base_ptr              as_base_ptr;
          decl_file_ptr         as_decl_file_ptr;
          declaration_ptr       as_declaration_ptr;
          type_specif_ptr       as_specif_ptr;
          typedef_name_ptr      as_typedef_name_ptr;
          typedef_ptr           as_typedef_ptr;
          struct_ptr            as_struct_ptr;
          decl_ptr              as_decl_ptr;
          declarator_ptr        as_declarator_ptr;
        }


//construct declaration for base class

base construct struct_base_class ::

  member {
    public:
        virtual yy_sym_ptr yy_symbol()          { return(NULL); }
        virtual const char *display_name()      { return(NULL); }
        virtual int        type()               { return(0); }
  };


//other construct declarations

construct struct_decl_file :: all

base { public struct_base_class }

constructor body {
        cout << "AST constructed" << endl;
}

constructor initializer {
  num_structures (num_struct),
  num_typedefs  (num_typedef),
  num_unions    (num_union),
  num_typedef_structs   (num_typedef_struct),
  num_typedef_unions    (num_typedef_union) }

member {
  public:
    int         num_structures;
    int         num_typedefs;
    int         num_unions;
    int         num_typedef_structs;
```

```
    int         num_typedef_unions;
};


construct declaration :: all

base { public struct_base_class }

constructor initializer {
  decl_type (class_type) }

constructor body {
  switch (class_type) {
    case STRUCT_TYPE          : ++num_struct; break;
    case UNION_TYPE           : ++num_union; break;
    case TYPEDEF_STRUCT_TYPE  : ++num_typedef_struct; break;
    case TYPEDEF_STRUCT_TYPE2 : ++num_typedef_struct; ++num_struct; break;
    case TYPEDEF_UNION_TYPE   : ++num_typedef_union; break;
    case TYPEDEF_UNION_TYPE2  : ++num_typedef_union; ++num_union; break;
    case TYPEDEF_TYPE         : ++num_typedef; break;
  }
}

member {
  public:
    int         decl_type;
    int         type() { return(decl_type); }
};

construct STORAGE_SPECIF :: default

base { public struct_base_class }

constructor initializer {
  name (ident_ptr) }

member {
  public:
    char        *name;
    const char  *display_name() { return(name); }
};

construct TYPE_SPECIF :: all

base { public struct_base_class }

constructor initializer {
  name (ident_ptr) }

member {
  public:
    char        *name;
    const char  *display_name() { return(name); }
};

construct TYPE_QUALIF :: default
```

```
base { public struct_base_class }

constructor initializer {
  name (ident_ptr) }

member {
  public:
    char        *name;
    const char  *display_name() { return(name); }
};

construct INIT_DECLARATOR :: default

base { public struct_base_class }

constructor initializer {
  num_stars     (num_star),
  is_array      (array),
  array_size    (size_ptr),
  name          (name_ptr),
  MPI_Construct         (0),
  AutoMap_CTpUsed       (ctpused),
  AutoMap_ATpUsed       (atpused),
  AutoMap_HTpUsed       (htpused) }

member {
  public:
    int         num_stars;
    int         is_array;
    int         MPI_Construct; // 1 if the MPI_Datatype already declared
                               // 2 if the MPI_Datatype already defined
    int         AutoMap_CTpUsed;        // 1 if used as CTp
    int         AutoMap_ATpUsed;        // 1 if used as ATp
    int         AutoMap_HTpUsed;        // 1 if used as HTp
    char        *array_size;
    char        *name;
    const char  *display_name() { return(name); }
};

construct TYPEDEF_NAME :: default

base {public struct_base_class }

constructor initializer {
  name (name_ptr) }

member {
  public:
    char        *name;
    const char  *display_name() { return(name); }
};

construct struct_or_union_specif :: all

base { public struct_base_class }

constructor initializer {
```

```
  name (struct_name),
  num_fields (num_var),
  num_pointers (0),
  num_decls (num_decl) }

member {
  public:
    char        *name;
    int         num_fields;
    int         num_pointers;
    int         num_decls;
    const char  *display_name() { return(name); }
};

construct enum_specifier :: default

base { public struct_base_class }
;

construct STRUCT_DECL :: all

base { public struct_base_class }

constructor initializer {
  num_declarators (num_declarator) }

constructor body {
  ++num_decl;
}

member {
  public:
    int         num_declarators;
};

construct STRUCT_DECLARATOR :: default

base { public struct_base_class }

constructor initializer {
  num_stars (num_star),
  AutoMap_Ptr (0),  // 1 if you want to send what the field points to
  is_array (array),
  array_size (size_ptr),
  name (name_ptr) }

member {
  public:
    int         num_stars;
    int         AutoMap_Ptr;
    int         is_array;
    char        *array_size;
    char        *name;
    const char  *display_name() { return(name); }
};
```

```
////////////////////////////////////////////////////////////////////////////////
// Beginning of the parser rules                                              //
////////////////////////////////////////////////////////////////////////////////

start struct_decl_file;

struct_decl_file  : { num_struct=0; num_union=0; num_typedef=0;
                      num_typedef_struct=0; num_typedef_union=0;
                      to_quit_psr=0; }
                   ( declaration )*
                  ;

declaration  :  { class_type=0; ctpused=0; atpused=0; htpused=0;
                  name_ptr=NULL; read_struct=0; }
                ( STORAGE_SPECIF | TYPE_QUALIF )*
                ( TYPEDEF_NAME | TYPE_SPECIF )
                ( STORAGE_SPECIF | TYPE_QUALIF | TYPE_SPECIF )*
                ( INIT_DECLARATOR  (COMMA  INIT_DECLARATOR)* )?
                SEMI
             ;

STORAGE_SPECIF :  auto | register | static | extern
               |  typedef  { class_type = TYPEDEF_TYPE; }
               ;

TYPE_SPECIF  :  char | int | float | void | short | long | double | signed
             |  unsigned
             |  struct_or_union_specif
             |  enum_specifier
             ;

TYPE_QUALIF  :  const | volatile
             ;

INIT_DECLARATOR :  DECLARATOR ( EQUAL INITIALIZER )?
                ;

struct_or_union_specif :
                struct_or_union { struct_name=NULL; read_struct=1; }
              ( IDENTIFIER ?
                     LBRACE  { num_var=0; num_decl=0; read_struct=0; }
                    ( STRUCT_DECL )+
                     RBRACE
        { if (class_type == TYPEDEF_STRUCT_TYPE)
            class_type=TYPEDEF_STRUCT_TYPE2;
          if (class_type == TYPEDEF_UNION_TYPE)
            class_type=TYPEDEF_UNION_TYPE2; }
              | IDENTIFIER
        { if (class_type == STRUCT_TYPE || class_type == UNION_TYPE)
            class_type=0; } )
             ;

struct_or_union : struct
        { if (class_type == TYPEDEF_TYPE) class_type=TYPEDEF_STRUCT_TYPE;
          else if (class_type == 0) class_type=STRUCT_TYPE; }
                | union
        { if (class_type == TYPEDEF_TYPE) class_type=TYPEDEF_UNION_TYPE;
```

```
          else if (class_type == 0) class_type=UNION_TYPE; }
                ;

STRUCT_DECL  :  { num_declarator=0; }
                ( TYPE_QUALIF )*
                ( TYPEDEF_NAME | TYPE_SPECIF )
                ( TYPE_SPECIF | TYPE_QUALIF )*
                STRUCT_DECLARATOR  ( COMMA  STRUCT_DECLARATOR )*
                SEMI
             ;

STRUCT_DECLARATOR : DECLARATOR
                  | DECLARATOR ?  COLON  CST_EXPR
                  ;

DECLARATOR   :  { num_star=0; array=0; size_ptr = new char[2];
                  strcpy(size_ptr, "1"); }
                ( POINTER ) ?
                  DIRECT_DECL
             ;

DIRECT_DECL  :  IDENTIFIER
                        { ++num_var; ++num_declarator;
                          name_ptr=ident_ptr; }
             |  LPAR  DECLARATOR  RPAR
             |  DIRECT_DECL
                ( LBRACK  ( CST_EXPR  { strcat(size_ptr, "*");
                                        strcat(size_ptr, "(");
                                        strcat(size_ptr, ident_ptr);
                                        strcat(size_ptr, ")"); } )?  RBRACK
                  { array=1; }
                | LPAR  ( IDENTIFIER ( COMMA  IDENTIFIER )* ) ?  RPAR
                | LPAR  PARAMETER_DECL  ( COMMA  PARAMETER_DECL )*  RPAR )
             ;

POINTER      :  STAR { ++num_star; } ( TYPE_QUALIF | STAR { ++num_star; } )*
             ;

PARAMETER_DECL : ( STORAGE_SPECIF | TYPE_QUALIF )*
                 ( TYPEDEF_NAME | TYPE_SPECIF )
                 ( STORAGE_SPECIF | TYPE_QUALIF | TYPE_SPECIF )*
                 ( DECLARATOR | ABSTRACT_DECLARATOR ) ?
               ;

ABSTRACT_DECLARATOR : POINTER
                    | POINTER ?  DIRECT_ABST_DECL
                    ;

DIRECT_ABST_DECL    :  LPAR  ABSTRACT_DECLARATOR  RPAR
                    |  DIRECT_ABST_DECL ?
                       ( LBRACK  CST_EXPR ?  RBRACK
                       | LPAR  (PARAMETER_DECL (COMMA PARAMETER_DECL)* )? RPAR)
                    ;

TYPEDEF_NAME :  IDENTIFIER { name_ptr=ident_ptr; }
             ;
```

```
enum_specifier :  enum
                  IDENTIFIER ?
                    LBRACE
                    ENUMERATOR  ( COMMA  ENUMERATOR )*  RBRACE
               |  enum
                  IDENTIFIER
               ;

ENUMERATOR   :  IDENTIFIER  ( EQUAL CST_EXPR )?
             ;

CST_EXPR     :  INTEGER_CST
             |  IDENTIFIER
             ;

INITIALIZER  :  CST_EXPR
             |  LBRACE  INITIALIZER ( COMMA  INITIALIZER )*  COMMA ? RBRACE
             ;
```

Appendix G

AutoMap_main.cxx

```
////////////////////////////////////////////////////////////////////////////////
//
//  FILENAME:           AutoMap_main.cxx
//
//  FILE DESCRIPTION:   Main routine which
//                      Creates the objects and calls
//                      the lexers and parsers for AutoMap.
//                      Outputs the files userdefs.h, mpitypes.c
//                      and automap.h, using the AST.
//
//  version 1.1
//
//  Eric Baland         January 1997
////////////////////////////////////////////////////////////////////////////////

#include "yy_ansi.h"
#include "yy_stdio.h"
#include "yy_bool.h"
#include "yy_errst.h"
#include "yy_inpst.h"
#include "AM1_lex.h"
#include "AM2_lex.h"
#include "AM1_psr.h"
#include "AM2_psr.h"
#include "yy_sym.h"
#include "yy_inpfi.h"

#include <stdlib.h>
#include <iostream.h>
#include <string.h>
#include <ctype.h>

#include "yy_err.tbl"

#ifdef yy_has_synas_
#include "yy_syna.tbl"
#else
const yy_synamsg_dflt_ptr yy_syna_tbl = NULL;
```

```
const int yy_syna_tbl_max_ = 0;
#endif /* yy_has_synas_ */


int       num_ctp, num_atp, num_htp;
FILE      *mpitypes_ptr;
AM2_parser  *yy_parser;

void error_msg() {
  cout << "Please, put the input-file name as an argument.\n"
       << "Type AutoMap -help for more information."
       << endl;
}


void help_msg() {
  cout << "AutoMap Help:\n\n"
       << "Synopsis\n"
       << "  AutoMap filename [option]\n"
       << "Options\n"
       << "    -c   : for a C file\n"
       << "    -f77 : for a Fortran-77 file\n"
       << "    -f90 : for a Fortran-90 file\n"
       << "    -c++ : for a C++ file\n"
       << endl;
}

declaration_ptr
find_declaration (decl_file_ptr obj_ptr, int obj_type, int obj_rank)
     // Return a pointer to the declaration object of type "obj_type"
     // and of rank "obj_rank" in the root object pointed by "obj_ptr"
{
  int  i;
  int count=0;

  for (i=0; i<obj_ptr->yy_num_operands(); i++) {
    if (obj_ptr->yy_operand(i).as_declaration_ptr->type() == obj_type)
      if (++count == obj_rank)

        // return the object searched
        return(obj_ptr->yy_operand(i).as_declaration_ptr);
  }
  return(NULL); // object not found
}

type_specif_ptr find_specif (declaration_ptr obj_ptr, int obj_rank)
     // Return a pointer to the TYPE_SPECIF object of rank
     // obj_rank in the declaration pointed by obj_ptr
{
  int count=0, i=-1;

  while (count++ < obj_rank)
    while (obj_ptr->yy_operand(++i).as_base_ptr->yy_type() !=
           yy_parser->TYPE_SPECIF_)
      if (obj_ptr->yy_operand(i).as_base_ptr->yy_type() ==
          yy_parser->INIT_DECLARATOR_ || i >= obj_ptr->yy_num_operands()-2)
        return(NULL);
```

```
  return(obj_ptr->yy_operand(i).as_specif_ptr);
}

typedef_ptr find_typedef (declaration_ptr obj_ptr)
     // Return a pointer to the first INIT_DECLARATOR object
     // in the declaration pointed by obj_ptr
{
  int i;

  if (obj_ptr->type() == STRUCT_TYPE || obj_ptr->type() == UNION_TYPE ||
      obj_ptr->type() == 0)
    return(NULL);
  for (i=0; i<obj_ptr->yy_num_operands()-1; i++) {
    if (obj_ptr->yy_operand(i).as_base_ptr->yy_type() ==
        yy_parser->INIT_DECLARATOR_)
      return(obj_ptr->yy_operand(i).as_typedef_ptr);
  }
  return(NULL);
}

struct_ptr find_struct (declaration_ptr obj_ptr)
     // Return a pointer to the struct object contained in the
     // declaration pointed by obj_ptr
{
  if (obj_ptr->type() != STRUCT_TYPE &&
      obj_ptr->type() != TYPEDEF_STRUCT_TYPE &&
      obj_ptr->type() != TYPEDEF_STRUCT_TYPE2)
    return(NULL);
  return(find_specif(obj_ptr, 1)->yy_operand(0).as_struct_ptr);
}

decl_ptr find_decl (struct_ptr obj_ptr, int obj_rank)
     // Return a pointer to the STRUCT_DECL object
     // which contains the declarator of rank "obj_rank"
     // in the struct_or_union object pointed by "obj_ptr"
{
  int  i, j;
  int count=0;

  for (i=2+(obj_ptr->display_name() != NULL ? 1 : 0);
       i<obj_ptr->yy_num_operands()-1; i++) {
    if (obj_ptr->yy_operand(i).as_base_ptr->yy_type() ==
        yy_parser->STRUCT_DECL_)
      for (j=0; j<obj_ptr->yy_operand(i).as_decl_ptr->num_declarators; j++)
        if (++count == obj_rank)
          return(obj_ptr->yy_operand(i).as_decl_ptr);
  }
  return(NULL);
}

declarator_ptr find_declarator (decl_ptr obj_ptr, int obj_rank)
     // Return a pointer to the struct_declarator object
     // of rank "obj_rank" in the struct_decl object
     // pointed by "obj_ptr"
{
  int i;
  int count=0;
```

```
  for (i=1; i<obj_ptr->yy_num_operands()-1; i++) {
    if (obj_ptr->yy_operand(i).as_base_ptr->yy_type() ==
        yy_parser->STRUCT_DECLARATOR_)
      if (++count == obj_rank)
        return(obj_ptr->yy_operand(i).as_declarator_ptr);
      else
        ++i;
  }
  return(NULL);
}


decl_ptr find_decl (struct_ptr obj_ptr, int obj_rank, int obj_stars,
                   int automap_ptr)
     // Return a pointer to the STRUCT_DECL object
     // which contains the declarator of rank "obj_rank" having "obj_stars"
     // star(s)
     // in the struct_or_union object "obj_ptr" points to
{
  int i, j;
  int count=0;

  for (i=2+(obj_ptr->display_name() != NULL ? 1 : 0);
       i<obj_ptr->yy_num_operands()-1; i++) {
    if (obj_ptr->yy_operand(i).as_base_ptr->yy_type() ==
        yy_parser->STRUCT_DECL_)
      for (j=0; j<obj_ptr->yy_operand(i).as_decl_ptr->num_declarators; j++)
        if ((find_declarator(obj_ptr->yy_operand(i).as_decl_ptr, j+1)->
             num_stars == obj_stars) &&
            (find_declarator(obj_ptr->yy_operand(i).as_decl_ptr, j+1)->
             AutoMap_Ptr == automap_ptr))
          if (++count == obj_rank)
            return(obj_ptr->yy_operand(i).as_decl_ptr);
  }
  return(NULL);
}


declarator_ptr find_declarator (struct_ptr obj_ptr, int obj_rank)
     // Return a pointer to the declarator object
     // of rank "obj_rank" in the structure object pointed by "obj_ptr"
{
  int i, j;
  int count=0;

  for (i=2+(obj_ptr->display_name() != NULL ? 1 : 0);
       i<obj_ptr->yy_num_operands()-1; i++) {
    if (obj_ptr->yy_operand(i).as_base_ptr->yy_type() ==
        yy_parser->STRUCT_DECL_)
      for (j=0; j<obj_ptr->yy_operand(i).as_decl_ptr->num_declarators; j++)
        if (++count == obj_rank)
          return(find_declarator(obj_ptr->yy_operand(i).as_decl_ptr, j+1));
 }
 return(NULL);
}
```

```
declarator_ptr find_declarator(struct_ptr obj_ptr, int obj_rank, int obj_stars,
                               int automap_ptr)
     // Return a pointer to the declarator object
     // of rank "obj_rank" having "obj_stars" star(s)
     // in the struct_or_union object pointed by "obj_ptr"
{
  int i, j;
  int count=0;

  for (i=2+(obj_ptr->display_name() != NULL ? 1 : 0);
       i<obj_ptr->yy_num_operands()-1; i++) {
    if (obj_ptr->yy_operand(i).as_base_ptr->yy_type() ==
        yy_parser->STRUCT_DECL_)
      for (j=0; j<obj_ptr->yy_operand(i).as_decl_ptr->num_declarators; j++)
        if ((find_declarator(obj_ptr->yy_operand(i).as_decl_ptr, j+1)->
             num_stars == obj_stars) &&
            (find_declarator(obj_ptr->yy_operand(i).as_decl_ptr, j+1)->
             AutoMap_Ptr == automap_ptr))
          if (++count == obj_rank)
            return(find_declarator(obj_ptr->yy_operand(i).as_decl_ptr, j+1));
  }
  return(NULL);
}


type_specif_ptr find_specif (decl_ptr obj_ptr, int obj_rank)
     // Return a pointer to the TYPE_SPECIF object of rank
     // obj_rank in the STRUCT_DECL pointed by obj_ptr
{
  int count=0, i=-1;

  while (count++ < obj_rank)
    while (obj_ptr->yy_operand(++i).as_base_ptr->yy_type() !=
           yy_parser->TYPE_SPECIF_)
      if (obj_ptr->yy_operand(i).as_base_ptr->yy_type() ==
          yy_parser->STRUCT_DECLARATOR_)
        return(NULL);
  return(obj_ptr->yy_operand(i).as_specif_ptr);
}

typedef_name_ptr find_typedef_name (decl_ptr obj_ptr)
     // Return a pointer to the TYPEDEF_NAME object used in the STRUCT_DECL
     // pointed by obj_ptr
{
  int i=-1;
  base_ptr base_p;


  base_p = obj_ptr->yy_operand(0).as_base_ptr;
  if (base_p->yy_type() != yy_parser->TYPE_QUALIF_ &&
      base_p->yy_type() != yy_parser->TYPEDEF_NAME_)
    return(NULL);
  while (obj_ptr->yy_operand(++i).as_base_ptr->yy_type() ==
         yy_parser->TYPE_QUALIF_) ;
  if (obj_ptr->yy_operand(i).as_base_ptr->yy_type() ==
      yy_parser->TYPEDEF_NAME_)
```

```
    return(obj_ptr->yy_operand(i).as_typedef_name_ptr);
  return(NULL);
}

struct_ptr find_struct (decl_file_ptr obj_ptr, declaration_ptr type_ptr)
     // Return a pointer to the structure object
     // which the typedef pointed by type_ptr refers to
{
  int i;

  if (type_ptr->type() == TYPEDEF_STRUCT_TYPE2)
    return(find_struct(type_ptr));
  if (type_ptr->type() != TYPEDEF_STRUCT_TYPE)
    return(NULL);
  for (i=0; i<obj_ptr->yy_num_operands(); i++) {
    if (obj_ptr->yy_operand(i).as_declaration_ptr->type() == STRUCT_TYPE ||
        obj_ptr->yy_operand(i).as_declaration_ptr->type() ==
        TYPEDEF_STRUCT_TYPE2)
      if (find_struct(obj_ptr->yy_operand(i).as_declaration_ptr)->
                      display_name() != NULL &&
          find_struct(type_ptr)->display_name() != NULL)
        if (strcmp(find_struct(obj_ptr->yy_operand(i).as_declaration_ptr)->
                   display_name(), find_struct(type_ptr)->display_name()) == 0)
          return(find_struct(obj_ptr->yy_operand(i).as_declaration_ptr));
  }
  return(NULL);
}


declaration_ptr find_declaration_typedef (decl_file_ptr obj_ptr, int obj_rank,
                                          int obj_stars)
     // Return a pointer to the declaration object which contains
     // the typedef_struct of rank "obj_rank", having "obj_stars" star(s)
{
  int i;
  int count=0;

  for (i=0; i<obj_ptr->yy_num_operands(); i++) {
    if (obj_ptr->yy_operand(i).as_declaration_ptr->type()
        == TYPEDEF_STRUCT_TYPE ||
        obj_ptr->yy_operand(i).as_declaration_ptr->type()
        == TYPEDEF_STRUCT_TYPE2)
      if (find_typedef(obj_ptr->yy_operand(i).as_declaration_ptr)->num_stars
          == obj_stars)
        if (++count == obj_rank)
          return(obj_ptr->yy_operand(i).as_declaration_ptr);
  }
  return(NULL);
}

declaration_ptr find_declaration_struct (decl_file_ptr obj_ptr, int obj_rank)
     // Return a pointer to the declaration object which contains
     // the structure of rank "obj_rank"
{
  int i;
  int count=0;
```

```
  for (i=0; i<obj_ptr->yy_num_operands(); i++) {
    if (obj_ptr->yy_operand(i).as_declaration_ptr->type()
        == STRUCT_TYPE ||
        obj_ptr->yy_operand(i).as_declaration_ptr->type()
        == TYPEDEF_STRUCT_TYPE2)
      if (++count == obj_rank)
        return(obj_ptr->yy_operand(i).as_declaration_ptr);
  }
  return(NULL);
}

declaration_ptr find_CTp (decl_file_ptr obj_ptr, int obj_rank)
     // Return a pointer to the declaration object
     // which is the CTp of rank "obj_rank"
{
  int i;
  int count=0;
  typedef_ptr typedef_p;

  for (i=0; i<obj_ptr->yy_num_operands(); i++) {
    typedef_p=find_typedef(obj_ptr->yy_operand(i).as_declaration_ptr);
    if (obj_ptr->yy_operand(i).as_declaration_ptr->type()
        == TYPEDEF_STRUCT_TYPE ||
        obj_ptr->yy_operand(i).as_declaration_ptr->type()
        == TYPEDEF_STRUCT_TYPE2)
      if (typedef_p->num_stars == 0 &&
          typedef_p->is_array == 0 &&
          typedef_p->AutoMap_CTpUsed == 1)
        if (count++ == obj_rank)
          return(obj_ptr->yy_operand(i).as_declaration_ptr);
  }
  return(NULL);
}

declaration_ptr find_ATp (decl_file_ptr obj_ptr, int obj_rank)
     // Return a pointer to the declaration object
     // which is the ATp of rank "obj_rank"
{
  int i;
  int count=0;
  typedef_ptr  typedef_p;

  for (i=0; i<obj_ptr->yy_num_operands(); i++) {
    typedef_p=find_typedef(obj_ptr->yy_operand(i).as_declaration_ptr);
    if (obj_ptr->yy_operand(i).as_declaration_ptr->type()
        == TYPEDEF_STRUCT_TYPE ||
        obj_ptr->yy_operand(i).as_declaration_ptr->type()
        == TYPEDEF_STRUCT_TYPE2)
      if (typedef_p->num_stars == 0 &&
          typedef_p->is_array == 0 &&
          typedef_p->AutoMap_ATpUsed == 1)
        if (count++ == obj_rank)
          return(obj_ptr->yy_operand(i).as_declaration_ptr);
  }
  return(NULL);
}
```

```
declaration_ptr find_HTp (decl_file_ptr obj_ptr, int obj_rank)
     // Return a pointer to the declaration object
     // which is the HTp of rank "obj_rank"
{
  int i;
  int count=0;
  typedef_ptr  typedef_p;

  for (i=0; i<obj_ptr->yy_num_operands(); i++) {
    typedef_p=find_typedef(obj_ptr->yy_operand(i).as_declaration_ptr);
    if (obj_ptr->yy_operand(i).as_declaration_ptr->type()
        == TYPEDEF_STRUCT_TYPE ||
        obj_ptr->yy_operand(i).as_declaration_ptr->type()
        == TYPEDEF_STRUCT_TYPE2)
      if (typedef_p->num_stars == 0 &&
          typedef_p->is_array == 0 &&
          typedef_p->AutoMap_HTpUsed == 1)
        if (count++ == obj_rank)
          return(obj_ptr->yy_operand(i).as_declaration_ptr);
  }
  return(NULL);
}

declaration_ptr find_XTp (decl_file_ptr obj_ptr, int obj_rank)
     // Return a pointer to the declaration object
     // which is the XTp of rank "obj_rank"
{
  if (obj_rank < num_ctp)
    return(find_CTp(obj_ptr, obj_rank));
  else if (obj_rank < num_atp+num_ctp)
    return(find_ATp(obj_ptr, obj_rank-num_ctp));
  else
    return(find_HTp(obj_ptr, obj_rank-num_ctp-num_atp));
}

declaration_ptr find_declaration_typedef (decl_file_ptr obj_ptr,
                                          decl_ptr struct_decl_ptr)
     // Return a pointer to the declaration object which contains the typedef
     // which is used in the structure declaration "struct_decl_ptr" points to
{
  int i;

  if (find_typedef_name(struct_decl_ptr) == NULL) return(NULL);

  for (i=0; i<obj_ptr->yy_num_operands(); i++) {
    if (obj_ptr->yy_operand(i).as_declaration_ptr->type() != STRUCT_TYPE &&
        obj_ptr->yy_operand(i).as_declaration_ptr->type() != UNION_TYPE &&
        obj_ptr->yy_operand(i).as_declaration_ptr->type() != 0)
      if (find_typedef(obj_ptr->yy_operand(i).as_declaration_ptr) != NULL)
        if (strcmp(find_typedef(obj_ptr->yy_operand(i).as_declaration_ptr)->
                   display_name(), find_typedef_name(struct_decl_ptr)->
                   display_name()) == 0)
          return(obj_ptr->yy_operand(i).as_declaration_ptr);
  }
  return(NULL);
}
```

```
declaration_ptr find_declaration_typedef_struct (decl_file_ptr obj_ptr,
                                                 decl_ptr struct_decl_ptr)
     // Return a pointer to the declaration object which contains
     // the typedef_struct
     // which is used in the structure declaration "struct_decl_ptr" points to
{
  int i;

  if (find_typedef_name(struct_decl_ptr) == NULL)
    return(NULL);
  for (i=0; i<obj_ptr->yy_num_operands(); i++) {
    if (obj_ptr->yy_operand(i).as_declaration_ptr->type()
        == TYPEDEF_STRUCT_TYPE ||
        obj_ptr->yy_operand(i).as_declaration_ptr->type()
        == TYPEDEF_STRUCT_TYPE2)
      if (strcmp(find_typedef(obj_ptr->yy_operand(i).as_declaration_ptr)->
                 display_name(), find_typedef_name(struct_decl_ptr)->
                 display_name()) == 0)
        return(obj_ptr->yy_operand(i).as_declaration_ptr);
  }
  return(NULL);
}


int find_ctp (decl_file_ptr obj_ptr, decl_ptr struct_decl_ptr)
     // Find the CTp number
     // of the type used in the structure field "struct_decl_ptr" points to
{
  int i;
  struct_ptr struct_p;

  if (find_typedef_name(struct_decl_ptr) == NULL)
    return(-1);
  if (find_declaration_typedef_struct(obj_ptr, struct_decl_ptr) == NULL)
    return(-1);
  struct_p = find_struct(obj_ptr,
                  find_declaration_typedef_struct(obj_ptr, struct_decl_ptr));
  for (i=0; i<num_ctp; i++) {
    if (find_struct(obj_ptr, find_CTp(obj_ptr, i))->display_name() != NULL &&
        struct_p->display_name() != NULL) {
      if (strcmp(find_struct(obj_ptr, find_CTp(obj_ptr, i))->display_name(),
                 struct_p->display_name()) == 0)
        return(i); }
    else
      if (strcmp(find_typedef(find_CTp(obj_ptr, i))->display_name(),
                 find_typedef_name(struct_decl_ptr)->display_name()) == 0)
        return(i);
  }
  return(-1);
}

int find_atp (decl_file_ptr obj_ptr, decl_ptr struct_decl_ptr)
     // Find the ATp number
     // of the type used in the structure field "struct_decl_ptr" points to
{
  int i;
```

```
  struct_ptr struct_p;

  if (find_typedef_name(struct_decl_ptr) == NULL)
    return(-1);
  if (find_declaration_typedef_struct(obj_ptr, struct_decl_ptr) == NULL)
    return(-1);
  struct_p = find_struct(obj_ptr,
                    find_declaration_typedef_struct(obj_ptr, struct_decl_ptr));
  for (i=0; i<num_atp; i++) {
    if (find_struct(obj_ptr, find_ATp(obj_ptr, i))->display_name() != NULL &&
        struct_p->display_name() != NULL) {
      if (strcmp(find_struct(obj_ptr, find_ATp(obj_ptr, i))->display_name(),
                 struct_p->display_name()) == 0)
        return(i); }
    else
      if (strcmp(find_typedef(find_ATp(obj_ptr, i))->display_name(),
                 find_typedef_name(struct_decl_ptr)->display_name()) == 0)
        return(i);
  }
  return(-1);
}

int find_htp (decl_file_ptr obj_ptr, decl_ptr struct_decl_ptr)
     // Find the HTp number
     // of the type used in the structure field "struct_decl_ptr" points to
{
  int i;
  struct_ptr struct_p;

  if (find_typedef_name(struct_decl_ptr) == NULL)
    return(-1);
  if (find_declaration_typedef_struct(obj_ptr, struct_decl_ptr) == NULL)
    return(-1);
  struct_p = find_struct(obj_ptr,
                    find_declaration_typedef_struct(obj_ptr, struct_decl_ptr));
  for (i=0; i<num_htp; i++) {
    if (find_struct(obj_ptr, find_HTp(obj_ptr, i))->display_name() != NULL &&
        struct_p->display_name() != NULL) {
      if (strcmp(find_struct(obj_ptr, find_HTp(obj_ptr, i))->display_name(),
                 struct_p->display_name()) == 0)
        return(i); }
    else
      if (strcmp(find_typedef(find_HTp(obj_ptr, i))->display_name(),
                 find_typedef_name(struct_decl_ptr)->display_name()) == 0)
        return(i);
  }
  return(-1);
}

declaration_ptr find_declaration_typedef_union (decl_file_ptr obj_ptr,
                                                const char *type_name)
     // Return a pointer to the declaration object which contains
     // the typedef_union which typedef name is type_name
{
  int i;

  for (i=0; i<obj_ptr->yy_num_operands(); i++)
```

```
    if (obj_ptr->yy_operand(i).as_declaration_ptr->type()
        == TYPEDEF_UNION_TYPE ||
        obj_ptr->yy_operand(i).as_declaration_ptr->type()
        == TYPEDEF_UNION_TYPE2)
      if (strcmp(find_typedef(obj_ptr->yy_operand(i).as_declaration_ptr)->
                 display_name(), type_name) == 0)
        return(obj_ptr->yy_operand(i).as_declaration_ptr);
  return(NULL);
}


char *find_MPI_Type (decl_file_ptr obj_ptr, const char *type_name)
     // Find the MPI_Type corresponding to the C type which name is "type_name"
{
  char *prefix;

  prefix = new char[9];
  strcpy (prefix, "AutoMap_");

  if (strcmp(type_name, "char") == 0) return("MPI_CHAR");
  if (strcmp(type_name, "int") == 0) return("MPI_INT");
  if (strcmp(type_name, "float") == 0) return("MPI_FLOAT");
  if (strcmp(type_name, "short") == 0) return("MPI_SHORT");
  if (strcmp(type_name, "long") == 0) return("MPI_LONG");
  if (strcmp(type_name, "double") == 0) return("MPI_DOUBLE");

  if (find_declaration_typedef_union(obj_ptr, type_name) != NULL)
    return("MPI_BYTE");

  return(strcat(prefix, type_name));
}

char *find_MPI_Type (decl_file_ptr obj_ptr, decl_ptr decl_p)
     // Find the MPI_Type corresponding to the C type
     // which is used in the struct_declaration pointed by decl_p
{
  if (find_typedef_name(decl_p) != NULL)
    return(find_MPI_Type(obj_ptr, find_typedef_name(decl_p)->display_name()));
  else if (find_specif(decl_p, 2) == NULL)
    return(find_MPI_Type(obj_ptr,
                         find_specif(decl_p, 1)->display_name()));
  else if (strcmp(find_specif(decl_p, 1)->display_name(), "unsigned") == 0) {
    if (strcmp(find_specif(decl_p, 2)->display_name(), "char") == 0)
      return("MPI_UNSIGNED_CHAR");
    else if (strcmp(find_specif(decl_p, 2)->display_name(), "int") == 0)
      return("MPI_UNSIGNED");
    else if (strcmp(find_specif(decl_p, 2)->display_name(), "short") == 0)
      return("MPI_UNSIGNED_SHORT");
    else if (strcmp(find_specif(decl_p, 2)->display_name(), "long") == 0)
      return("MPI_UNSIGNED_LONG");
  }
  else if (strcmp(find_specif(decl_p, 1)->display_name(), "long") == 0 &&
           strcmp(find_specif(decl_p, 2)->display_name(), "double") == 0)
    return("MPI_LONG_DOUBLE");
  else if (strcmp(find_specif(decl_p, 1)->display_name(), "long") == 0)
    return("MPI_LONG");
```

```
  return(find_MPI_Type(obj_ptr,
                       find_specif(decl_p, 2)->display_name()));
}

#ifndef STANDALONE
void AutoLink_Cfg(decl_file_ptr obj_ptr)
     // Configuration routine for AutoLink
{
    ///////////////////////////////////////////////////////////////////////////
    // Configuration questions
    ///////////////////////////////////////////////////////////////////////////

/*
This is configuration program for AutoLink. It will run AutoMap-AL,
open the alconfig.txt file and ask the user questions, that need to
be anwsered in order to properly configure AutoLink.
*/

/* Version 1.0 */

/* DEFINES */
#define MaxXTpUsed 10
#define MaxNameLength  50

int Selection[MaxXTpUsed];
int SelectCnt;
int FieldFunc;
int i, j, TypeCount;
char c;
char Name[MaxNameLength];
char buffer[200];

struct_ptr       structure;
typedef_ptr  typedef_p;


TypeCount = num_ctp+num_atp+num_htp;

printf(
"\n\n\n\n\n-----------------------------------------------------------------\n"
);
printf(
"--------   This is the AutoLink Configuration program   ---------\n"
);
printf(
"-----------------------------------------------------------------\n"
);

printf(
"\n\n\n\n\n---   Traversing Graphs  ---\n"
);
printf("\nDo you intend to use the AutoLink-traversing routine? (Y/N) ->");
if (gets(buffer) == NULL) printf("What'swrong\n");
sscanf(buffer, "%1s", &c);
if ((c == 'Y') || (c == 'y')) {

  /* MARK FIELDS */
```

```
printf("\nInstances of which of your datatypes might be 'reached' \n");
printf("from several different parents\n\n");

for (i=0; i<TypeCount; i++)
  printf("\t%d: %s\n",i, find_typedef(find_XTp(obj_ptr, i))->display_name());
printf("\nPlease enter their numbers, seperated by spaces:\n->");
if (gets(buffer) == NULL) printf("What's wrong\n");
SelectCnt = sscanf(buffer, "%d%d%d%d%d%d%d%d%d%d", &Selection[0],
                 &Selection[1], &Selection[2], &Selection[3], &Selection[4],
                 &Selection[5], &Selection[6], &Selection[7], &Selection[8],
                 &Selection[9]);

if (SelectCnt>0)
  printf(
"\nPlease enter the names of the fields, that may be used by AutoLink.\n\n");

for (i=0; i<SelectCnt; i++)
  {
    printf("%20s: ",
           find_typedef(find_XTp(obj_ptr, Selection[i]))->display_name());
    if (gets(buffer) == NULL) printf("What's wrong\n");
    sscanf(buffer,"%s", Name);
    fprintf(header_ptr, "\n#define mark");
    if (Selection[i]>=num_ctp+num_atp)
      fprintf(header_ptr, "H%d", Selection[i]-num_ctp-num_atp);
    else if (Selection[i]>=num_ctp)
      fprintf(header_ptr, "A%d", Selection[i]-num_ctp);
    else fprintf(header_ptr, "%d", Selection[i]);
    fprintf(header_ptr, "\t\t%s\n", Name);
  }


/* ARRAY SIZES */

printf(
       "\n\n\n\n\n---   Traversing Arrays and Strings  ---\n"
       );
for (i=0; i<TypeCount; i++) {

  typedef_p = find_typedef(find_XTp(obj_ptr, i));

  // find the structure which correspond to the type #i
  structure = find_struct(obj_ptr, find_XTp(obj_ptr, i));

  for (j=0; j<structure->num_pointers; j++) {

    if (find_atp(obj_ptr,
                 find_decl(structure, j+1, 1, 1)) != -1) {
      // if the pointer points to an ATp

      typedef_p = find_typedef(find_ATp(obj_ptr, find_atp(obj_ptr,
                  find_decl(structure, j+1, 1, 1))));

      printf(
         "\n\nThe %s-datatype has a pointer %s to the string-datatype %s.\n",
         find_typedef(find_XTp(obj_ptr, i))->display_name(),
```

```
         find_declarator(structure, j+1, 1, 1)->display_name(),
         typedef_p->display_name());
      printf(
"AutoLink can only traverse from a %s-instance \ninto a linked %s-string,\n",
find_typedef(find_XTp(obj_ptr, i))->display_name(),
typedef_p->display_name());
      printf("if the %s-instance can provide the string-length!\n",
             find_typedef(find_XTp(obj_ptr, i))->display_name());
      printf("Is this possible and should this be done (Y/N) ->");
      if (gets(buffer) == NULL) printf("What's wrong\n");
      sscanf(buffer, "%1s", &c);
      if ((c == 'Y') || (c == 'y'))
        {
          printf("\nIs the string-length\n");
          printf("   1: stored in a field, or is it\n");
          printf("   2: returned by a function\n->");
          if (gets(buffer) == NULL) printf("What's wrong\n");
          sscanf(buffer, "%1s", &c);
          if (c == '1') FieldFunc = 1; else FieldFunc = 0;
          printf("Please enter the name of that field or function: ");
          if (gets(buffer) == NULL) printf("What's wrong\n");
          sscanf(buffer,"%s", Name);
          if (FieldFunc == 1) fprintf(header_ptr, "#define ArrCnt%d%d",
         (i>=num_ctp+num_atp?i-num_ctp-num_atp:(i>=num_ctp?i-num_ctp:i)), j);
          else fprintf(header_ptr, "#define ArrCntF%d%d",
         (i>=num_ctp+num_atp?i-num_ctp-num_atp:(i>=num_ctp?i-num_ctp:i)), j);
          fprintf(header_ptr, "\t%s\n", Name);
          printf("\n\n\n\n");
        }
    }
    if (find_htp(obj_ptr,
                 find_decl(structure, j+1, 1, 1)) != -1) {
      // if the pointer points to a HTp

      typedef_p = find_typedef(find_HTp(obj_ptr, find_htp(obj_ptr,
                  find_decl(structure, j+1, 1, 1))));

      printf(
          "\n\nThe %s-datatype has a pointer %s to the array-datatype %s.\n",
          find_typedef(find_XTp(obj_ptr, i))->display_name(),
          find_declarator(structure, j+1, 1, 1)->display_name(),
          typedef_p->display_name());
      printf(
 "AutoLink can only traverse from a %s-instance \ninto a linked %s-array,\n",
 find_typedef(find_XTp(obj_ptr, i))->display_name(),
 typedef_p->display_name());
      printf("if the %s-instance can provide the array-size!\n",
             find_typedef(find_XTp(obj_ptr, i))->display_name());
      printf("Is this possible and should this be done (Y/N) ->");
      if (gets(buffer) == NULL) printf("What's wrong\n");
      sscanf(buffer, "%1s", &c);
      if ((c == 'Y') || (c == 'y'))
        {
          printf("\nIs the array-size\n");
          printf("   1: stored in a field, or is it\n");
          printf("   2: returned by a function\n->");
          if (gets(buffer) == NULL) printf("What's wrong\n");
```

```
            sscanf(buffer, "%1s", &c);
            if (c == '1') FieldFunc = 1; else FieldFunc = 0;
            printf("Please enter the name of that field or function: ");
            if (gets(buffer) == NULL) printf("What's wrong\n");
            sscanf(buffer,"%s", Name);

            if (FieldFunc ==1) fprintf(header_ptr, "#define HybCnt%d%d",
           (i>=num_ctp+num_atp?i-num_ctp-num_atp:(i>=num_ctp?i-num_ctp:i)), j);
            else fprintf(header_ptr, "#define HybCntF%d%d",
           (i>=num_ctp+num_atp?i-num_ctp-num_atp:(i>=num_ctp?i-num_ctp:i)), j);
            fprintf(header_ptr, "\t%s\n", Name);
            printf("\n\n\n\n");
          }
        printf("\n\n\n\n");
      }
    }
  }
}

/* CREATOR ROUTINES */

printf(
       "\n\n\n\n\n---   Memory Management   ---\n"
       );
printf(
"\nDo you wish that AutoLink uses your own memory management routines? (Y/N)->"
);
if (gets(buffer) == NULL) printf("What's wrong\n");
sscanf(buffer, "%1s", &c);
if ((c == 'Y') || (c == 'y')) {
  printf("\nPlease enter the names of the instance allocating routines \n");
  printf("for each datatype, or ENTER for none:\n");

  for (i=0; i<TypeCount; i++)
    {
      printf("%20s: ", find_typedef(find_XTp(obj_ptr, i))->display_name());
      if (gets(buffer) == NULL) printf("What's wrong\n");
      sscanf(buffer,"%s", Name);
      if (strlen(buffer)>0)
        {
          if (i<num_ctp)
            fprintf(header_ptr, "#define GetCTp%d \t%s\n", i, Name);
          else if (i<num_ctp+num_atp)
            fprintf(header_ptr, "#define GetATp%d \t%s\n", i-num_ctp, Name);
          else
            fprintf(header_ptr, "#define GetHTp%d \t%s\n",
                    i-num_ctp-num_atp, Name);
        }
    }
}
printf("\nConfiguration of AutoLink complete.\n\n\n");
}
#endif

////////////////////////////////////////////////////////////////////////////////
// Types construction
////////////////////////////////////////////////////////////////////////////////
```

```
void Types_construct(decl_file_ptr obj_ptr, int type, int num)
     // Write in mpitypes.c the code to create the MPI Types
     // type is 1 for CTp, 2 for ATp or 3 for HTp
     // num is the number of types to construct (num_ctp, num_atp...)
{
  int i, j;
  struct_ptr structure;
  typedef_ptr typedef_p;
  declaration_ptr declaration_p;

    for (i=0; i<num; i++) {
      // find the typedef which correspond to the type #i
      if (type == 1)
        declaration_p = find_CTp(obj_ptr,i);
      else if (type == 2)
        declaration_p = find_ATp(obj_ptr,i);
      else
        declaration_p = find_HTp(obj_ptr,i);
      typedef_p = find_typedef(declaration_p);
#ifdef STANDALONE
      fprintf(mpitypes_ptr, "\n/* %s */\n", typedef_p->display_name());
#else
      if (type == 1)
        fprintf(mpitypes_ptr, "\n/* CTp%d */\n", i);
      else if (type == 2)
        fprintf(mpitypes_ptr, "\n/* ATp%d */\n", i);
      else
        fprintf(mpitypes_ptr, "\n/* HTp%d */\n", i);
#endif

      // find the structure which correspond to the type #i
      structure = find_struct(obj_ptr, declaration_p);

      // Loop for the displacements
      for (j=0; j<structure->num_fields; j++) {
        fprintf(mpitypes_ptr, "MPI_Address(");
        if (!find_declarator(structure, j+1)->is_array)
          // if this field is not an array, print "&"
          fprintf(mpitypes_ptr, "&");
        if (type == 1)
          fprintf(mpitypes_ptr, "C");
        else if (type == 2)
          fprintf(mpitypes_ptr, "A");
        else
          fprintf(mpitypes_ptr, "H");
        fprintf(mpitypes_ptr, "Type%d.%s,\t&disp[%d]);\n", i,
                find_declarator(structure, j+1)->display_name(), j);
      }
      // Loop for the types
      for (j=0; j<structure->num_fields; j++) {
        if (find_declarator(structure, j+1)->num_stars == 0)
          fprintf(mpitypes_ptr, "type[%d] = %s;\n", j,
                  find_MPI_Type(obj_ptr, find_decl(structure, j+1)));
        else
          fprintf(mpitypes_ptr, "type[%d] = MPI_INT;\n", j);
      }
```

```
      // Loop for the blocklengths
      for (j=0; j<structure->num_fields; j++) {
        fprintf(mpitypes_ptr, "blocklen[%d] = ", j);
        if (find_declarator(structure, j+1)->is_array) {
          fprintf(mpitypes_ptr, "%s;\n",
                  find_declarator(structure, j+1)->array_size);
          if (find_typedef_name(find_decl(structure, j+1)) != NULL)
            if (find_declaration_typedef_union(obj_ptr,
                  find_typedef_name(find_decl(structure, j+1))->display_name())
                != NULL)
              fprintf(mpitypes_ptr, "*sizeof(%s);\n",
                 find_typedef_name(find_decl(structure, j+1))->display_name());
        }
        else if (find_typedef_name(find_decl(structure, j+1)) != NULL)
          if (find_declaration_typedef_union(obj_ptr,
                  find_typedef_name(find_decl(structure, j+1))->display_name())
              != NULL)
            fprintf(mpitypes_ptr, "sizeof(%s);\n",
               find_typedef_name(find_decl(structure, j+1))->display_name());
          else fprintf(mpitypes_ptr, "1;\n");
        else
          fprintf(mpitypes_ptr, "1;\n");
      }

      fprintf(mpitypes_ptr,
              "base = disp[0]; for(i=0; i<%d; i++) disp[i]-=base;\n",
              structure->num_fields);

#ifdef STANDALONE

      fprintf(mpitypes_ptr,
              "MPI_Type_struct(%d, blocklen, disp, type, &%s);\n",
              structure->num_fields,
              find_MPI_Type(obj_ptr, typedef_p->display_name()));
      fprintf(mpitypes_ptr, "MPI_Type_commit(&%s);\n",
              find_MPI_Type(obj_ptr, typedef_p->display_name()));

#else

      fprintf(mpitypes_ptr,
              "MPI_Type_struct(%d, blocklen, disp, type, &MPI_Types[%d]);\n",
              structure->num_fields, i+(type >= 2 ? num_ctp : 0)+
              (type == 3 ? num_atp : 0));
      fprintf(mpitypes_ptr, "MPI_Type_commit(&MPI_Types[%d]);\n", i+
              (type >= 2 ? num_ctp : 0)+(type == 3 ? num_atp : 0));

#endif
    }
}

void Types_declare(decl_file_ptr obj_ptr, declaration_ptr declaration_p)
     // Write in mpitypes.c the code to declare the MPI Type
     // corresponding to the typedef declaration pointed by declaration_p
{
  static int count=0;
  struct_ptr structure;
  typedef_ptr typedef_p;
```

```
  int i;

  if ((typedef_p = find_typedef(declaration_p)) == NULL)
    return;
  typedef_p->MPI_Construct = 1;

  //find the structure corresponding to the typedef
  structure = find_struct(obj_ptr, declaration_p);
  if (structure == NULL) return;

  for (i=0; i<structure->num_fields; i++)
    if (find_typedef_name(find_decl(structure, i+1)) != NULL &&
        find_declarator(structure, i+1)->num_stars == 0)

      if (find_typedef(find_declaration_typedef(obj_ptr, find_decl(structure,
           i+1)))->MPI_Construct == 0)
        Types_declare(obj_ptr,
                      find_declaration_typedef(obj_ptr, find_decl(structure,
                                                                  i+1)));

  fprintf(mpitypes_ptr, "%s\tTpDef%d;\n", typedef_p->display_name(), count++);
  fprintf(mpitypes_ptr, "MPI_Datatype\t%s;\n",
          find_MPI_Type(obj_ptr, typedef_p->display_name()));
}


void Types_construct(decl_file_ptr obj_ptr, declaration_ptr declaration_p)
     // Write in mpitypes.c the code to create the MPI Types
     // corresponding to the typedef declaration pointed by declaration_p
{
  static int count=0;
  struct_ptr structure;
  typedef_ptr typedef_p;
  int        i;

  if ((typedef_p = find_typedef(declaration_p)) == NULL)
    return;

  typedef_p->MPI_Construct = 2;

  //find the structure corresponding to the typedef
  structure = find_struct(obj_ptr, declaration_p);
  if (structure == NULL) return;

  for (i=0; i<structure->num_fields; i++)
    if (find_typedef_name(find_decl(structure, i+1)) != NULL &&
        find_declarator(structure, i+1)->num_stars == 0)

      if (find_typedef(find_declaration_typedef(obj_ptr, find_decl(structure,
           i+1)))->MPI_Construct == 1)
        Types_construct(obj_ptr,
                        find_declaration_typedef(obj_ptr,
                                                 find_decl(structure, i+1)));

  // construct the MPI Datatype corresponding to the typedef
  fprintf(mpitypes_ptr, "\n/* %s */\n",
          typedef_p->display_name());
```

```
  // loop for the displacements
  for (i=0; i<structure->num_fields; i++) {
    fprintf(mpitypes_ptr, "MPI_Address(");
    if (!find_declarator(structure, i+1)->is_array)
      // if this field is an array, do not print "&"
      fprintf(mpitypes_ptr, "&");
    fprintf(mpitypes_ptr, "TpDef%d.%s,\t&disp[%d]);\n",
            count,
            find_declarator(structure,i+1)->display_name(), i);
  }

  // loop for the types
  for (i=0; i<structure->num_fields; i++) {
    if (find_declarator(structure, i+1)->num_stars == 0)
      fprintf(mpitypes_ptr, "type[%d] = %s;\n", i,
              find_MPI_Type(obj_ptr, find_decl(structure, i+1)));
    else
      fprintf(mpitypes_ptr, "type[%d] = MPI_INT;\n", i);
  }

  // loop for the blocklengths
  for (i=0; i<structure->num_fields; i++) {
    fprintf(mpitypes_ptr, "blocklen[%d] = ", i);
    if (find_declarator(structure, i+1)->is_array) {
      fprintf(mpitypes_ptr, "%s;\n",
              find_declarator(structure, i+1)->array_size);
      if (find_typedef_name(find_decl(structure, i+1)) != NULL)
        if (find_declaration_typedef_union(obj_ptr,
                  find_typedef_name(find_decl(structure, i+1))->display_name())
            != NULL)
          fprintf(mpitypes_ptr, "*sizeof(%s);\n",
                 find_typedef_name(find_decl(structure, i+1))->display_name());
    }
    else if (find_typedef_name(find_decl(structure, i+1)) != NULL)
      if (find_declaration_typedef_union(obj_ptr,
                  find_typedef_name(find_decl(structure, i+1))->display_name())
          != NULL)
        fprintf(mpitypes_ptr, "sizeof(%s);\n",
               find_typedef_name(find_decl(structure, i+1))->display_name());
      else fprintf(mpitypes_ptr, "1;\n");
    else
      fprintf(mpitypes_ptr, "1;\n");
  }

  fprintf(mpitypes_ptr,
          "base = disp[0]; for(i=0; i<%d; i++) disp[i]-=base;\n",
          structure->num_fields);

  fprintf(mpitypes_ptr,
          "MPI_Type_struct(%d, blocklen, disp, type, &%s);\n",
          structure->num_fields,
          find_MPI_Type(obj_ptr, typedef_p->display_name()));
  fprintf(mpitypes_ptr, "MPI_Type_commit(&%s);\n",
          find_MPI_Type(obj_ptr, typedef_p->display_name()));
  ++count;
}
```

```
///////////////////////////////////////////////////////////////////////////////
//  MAIN
///////////////////////////////////////////////////////////////////////////////
int main (int argc, char *argv[])
{
  int                 code, i, j, k
#ifndef STANDALONE
    , num_pointers
#endif
    ;
  int                 opt_position=0; // position of the option in argv
  char                option_str[5];
  int                 option;
  yy_err_dflt_ptr     yy_err1, yy_err2;
  yy_inp_stream_ptr   yy_inp1, yy_inp2;
  yy_symtab_dflt_ptr  yy_symtab1, yy_symtab2;
  AM1_lexer           *yy_lexer1;
  AM1_parser          *yy_parser1;
  AM2_lexer           *yy_lexer2;
  decl_file_ptr       root;           // root of the AST
  declaration_ptr     declaration_p;
  struct_ptr          structure;
  typedef_ptr         typedef_p;
  decl_ptr            decl_p;
  declarator_ptr      declarator_p;

#ifndef STANDALONE
  FILE                *automap_ptr;
#endif

  //  check if argc>=2

  if (argc < 2) {
    error_msg();
    exit(1);
  }

  //  check the options

  if (argc == 2) {
    if (strncmp(argv[1], "-", 1) == 0) {
      help_msg(); exit(1); }
    else
      opt_position=2; }
  else if (argc == 3) {
    if (strncmp(argv[1], "-", 1) == 0)
      opt_position=1;
    else
      opt_position=2; }
  else {
    error_msg();
    exit(1);
  }

  if (argc == 2)
```

```
    strncpy(option_str, "-c", 4);
  else
    strncpy(option_str, argv[opt_position], 4);

  ///////////////////////////////////////////////////////////////////////////
  // Choice of the option
  ///////////////////////////////////////////////////////////////////////////

if (strcmp(option_str, "-c") == 0)
  option=1; // C file

else if (strcmp(option_str, "-f77") == 0)
  option=2; // Fortran 77

else if (strcmp(option_str, "-f90") == 0)
  option=3; // Fortran 90

else if (strcmp(option_str, "-c++") == 0)
  option=4; // C++

else {
  error_msg();
  exit(1);
}

switch (option) {

case 1:  // C file

  cout << "-- AutoMap1.1 for C files --"
       << endl;
#ifdef STANDALONE
  cout << "-- Stand-alone version --"
       << endl;
#endif

  //  open the "userdefs.h" file
#ifndef STANDALONE
  if ((header_ptr=fopen("userdefs.h", "w")) == NULL) {
    cout << "Can't open file...\n" << flush;
    exit(1);
  }
#endif

  //create objects for lexer & parser #1

  //  create an error object, the default is to report errors to a C++ stream
  yy_err1 = new yy_err_dflt_obj(
                             yy_err_tbl, yy_err_tbl_max_,
                             yy_syna_tbl, yy_syna_tbl_max_,
                             &cout
                             );

  //  create an input object which processes input from a file
  yy_inp1 = new yy_inp_stream_obj(yy_err1);

  //  create a symbol table object, the default is hashed symbol lookups
```

```
  yy_symtab1 = new yy_symtab_dflt_obj;

  //  create a lexer object
  yy_lexer1 = new AM1_lexer(yy_inp1, yy_symtab1,
                               AM1_lexer::yy_lex_class_AutoStart_);

  //  create a parser object
  yy_parser1 = new AM1_parser(yy_lexer1,
                                AM1_parser::yy_psr_class_AutoStart_,
                                AM1_parser::start_);


  //create objects for lexer & parser #2

  //  create an error object, the default is to report errors to a C++ stream
  yy_err2 = new yy_err_dflt_obj(
                             yy_err_tbl, yy_err_tbl_max_,
                             yy_syna_tbl, yy_syna_tbl_max_,
                             &cout
                             );

  //  create an input object which processes input from a file
  yy_inp2 = new yy_inp_stream_obj(yy_err2);


  //  create a symbol table object, the default is hashed symbol lookups
  yy_symtab2 = new yy_symtab_dflt_obj;


  //  create a lexer object
  yy_lexer2 = new AM2_lexer(yy_inp2, yy_symtab2,
                               AM2_lexer::yy_lex_class_AutoMap_);


  //  create a parser object
  yy_parser = new AM2_parser(yy_lexer2,
                                AM2_parser::yy_psr_class_AutoMap_,
                                AM2_parser::struct_decl_file_);


  // begin reading from the file
  yy_inp1 -> yy_inp_is_fstream(argv[3-opt_position]);

  // lex and parse the input file
  code = yy_psr(yy_parser1);

  if (code == 0) {  // success
    cout << "First parse completed successfully."
         << endl;

    // begin mark detected: give the stream to the AutoMap grammar
    yy_inp2 -> yy_inp_is_opened_stream( yy_inp1->yy_inp_cur_stream() );

////////////////////////////////////////////////////////////////////////////////
// begin writing "userdefs.h"
////////////////////////////////////////////////////////////////////////////////
```

```
#ifndef STANDALONE

    fprintf(header_ptr, "#ifndef USERDEF\n#define USERDEF\n\n");

    fprintf(header_ptr, "/* TYPES */\n");

#endif

    code = yy_psr(yy_parser);

  if (code == 0) {  // success: file parsed successfully until the end mark
                    // AST constructed

    cout << "Main parse completed successfully."
         << endl;

    //  open the "mpitypes.c" file
    if ((mpitypes_ptr=fopen("mpitypes.c", "w")) == NULL) {
      cout << "Can't open file...\n" << flush;
      exit(1);
    }


    //  open the "automap.h" file
#ifndef STANDALONE
    if ((automap_ptr=fopen("automap.h", "w")) == NULL) {
      cout << "Can't open file...\n" << flush;
      exit(1);
    }
#endif

    // root of the AST
    root = yy_parser->yy_psr_ref(1).as_decl_file_ptr;

    ///////////////////////////////////////////////////////////////////////////
    // put the typedef struct declared as "AutoMap_CTpUsed" in CTp.(no pointer)
    // do the same for Atp and HTp
    ///////////////////////////////////////////////////////////////////////////

cout << "num_typedefs:\t"
     << root->num_typedefs << endl
     << "num_structures:\t"
     << root->num_structures << endl
     << "num_unions:\t"
     << root->num_unions << endl
     << "num_typedef_structs:\t"
     << root->num_typedef_structs << endl
     << "num_typedef_unions:\t"
     << root->num_typedef_unions << endl;


    for (i=0; i<root->num_typedef_structs; i++) {
      if (find_declaration_typedef(root, i+1, 0) == NULL) break;
      typedef_p = find_typedef(find_declaration_typedef(root, i+1, 0));

      if (typedef_p != NULL)
        if (typedef_p->AutoMap_CTpUsed == 1)
```

```
          ++num_ctp;
        else if (typedef_p->AutoMap_ATpUsed == 1)
          ++num_atp;
        else if (typedef_p->AutoMap_HTpUsed == 1)
          ++num_htp;
    }
cout << "num_ctp: " << num_ctp << endl << "num_atp: " << num_atp << endl
     << "num_htp: " << num_htp << endl;
#ifndef STANDALONE

    fprintf(header_ptr,
         "\n\n/* DEFINES */\n\n#define MaxFields\t50\n#define CTpUsed \t%d\n",
            num_ctp);
    fprintf(header_ptr,
            "#define ATpUsed \t%d\n#define HTpUsed \t%d\n\n",
            num_atp, num_htp);

#endif

    ///////////////////////////////////////////////////////////////////////////
    // correct num_stars and AutoMap_Ptr
    ///////////////////////////////////////////////////////////////////////////

    for (i=0; i<root->num_structures; i++) {
      structure = find_struct(find_declaration_struct(root, i+1));

      for (j=0; j<structure->num_decls; j++) {
        decl_p = find_decl(structure, j+1);

        if ((declaration_p = find_declaration_typedef(root, decl_p)) != NULL) {
          // increase num_stars of the field if the TYPEDEF_NAME is a pointer
          for(k=0; k<decl_p->num_declarators; k++) {
            declarator_p = find_declarator(decl_p, k+1);
            declarator_p->num_stars += find_typedef(declaration_p)->num_stars;

            // change the "is_array" field and adjust the array_size
            if (declarator_p->is_array == 0) {
              declarator_p->is_array = find_typedef(declaration_p)->is_array;
              strcpy(declarator_p->array_size,
                     find_typedef(declaration_p)->array_size);
            }
            else {
              strcat(declarator_p->array_size, "*");
              strcat(declarator_p->array_size,
                     find_typedef(declaration_p)->array_size);
            }
          }

          // put AutoMap_Ptr at 1 if the TYPEDEF_NAME is a CTp, ATp or HTp
          // and the field a pointer to this type

          if ((find_ctp(root, decl_p) != -1) ||
              (find_atp(root, decl_p) != -1) ||
              (find_htp(root, decl_p) != -1))
            for(k=0; k < decl_p->num_declarators; k++) {
              declarator_p = find_declarator(decl_p, k+1);
              if (declarator_p->num_stars == 1)
```

```
            declarator_p->AutoMap_Ptr = 1; }
        }
      }
    }

    ////////////////////////////////////////////////////////////////////////////
    // correct num_pointers to fit the number of AutoMap_Ptrs
    ////////////////////////////////////////////////////////////////////////////

    for (i=0; i<root->num_structures; i++) {
      structure = find_struct(find_declaration_struct(root, i+1));
      structure->num_pointers = 0;

      for (j=0; j<structure->num_fields; j++) {
        if (find_declarator(structure, j+1)->AutoMap_Ptr != 0)
          structure->num_pointers++;
      }
    }



#ifndef STANDALONE

    ////////////////////////////////////////////////////////////////////////////
    // loop for the typedef...
    ////////////////////////////////////////////////////////////////////////////
    for (i=0; i<num_ctp+num_atp+num_htp; i++) {
      // find the typedef corresponding to the XTp#i
      typedef_p = find_typedef(find_XTp(root, i));

      fprintf(header_ptr, "typedef %s\t", typedef_p->display_name());
      fprintf(header_ptr, i<num_ctp?"C":(i<num_ctp+num_atp?"A":"H"));
      fprintf(header_ptr, "Tp%d;",
              i<num_ctp?i:(i<num_ctp+num_atp?i-num_ctp:i-num_ctp-num_atp));
      fprintf(header_ptr, "\t/* datatype number %d */\n", i);
    }
    fprintf(header_ptr, "\n");

    ////////////////////////////////////////////////////////////////////////////
    // loop for the #define CT, AT and HT.
    ////////////////////////////////////////////////////////////////////////////
    for (i=0; i<num_ctp; i++)
      fprintf(header_ptr, "#define\tCT%d\n", i);
    for (i=0; i<num_atp; i++)
      fprintf(header_ptr, "#define\tAT%d\n", i);
    for (i=0; i<num_htp; i++)
      fprintf(header_ptr, "#define\tHT%d\n", i);

    ////////////////////////////////////////////////////////////////////////////
    // loop for the #define Ptr, Arr and Hyb
    ////////////////////////////////////////////////////////////////////////////
    for (i=0; i<num_ctp+num_atp+num_htp; i++) {
      //find the typedef corresponding to the XTp#i
      typedef_p = find_typedef(find_XTp(root, i));

      // find the structure which correspond to the XTp #i
      structure = find_struct(root, find_XTp(root, i));
```

```
  fprintf(header_ptr, "\n/* %s */\n",
          typedef_p->display_name());

  num_pointers = structure->num_pointers;
  for (j=0; j<num_pointers; j++) {
    decl_p = find_decl(structure, j+1, 1, 1);

    fprintf(header_ptr, "#define ");

    if (find_ctp(root, decl_p) != -1)
      fprintf(header_ptr, "PtrFld");
    else if (find_atp(root, decl_p) != -1)
      fprintf(header_ptr, "ArrFld");
    else
      fprintf(header_ptr, "HybFld");

    if (i >= num_ctp+num_atp)
      fprintf(header_ptr, "H%d", i-num_ctp-num_atp);
    else if (i >= num_ctp)
      fprintf(header_ptr, "A%d", i-num_ctp);
    else
      fprintf(header_ptr, "%d", i);
    fprintf(header_ptr, "%d\t%s\n", j,
            find_declarator(structure, j+1, 1, 1)->display_name());

    fprintf(header_ptr, "#define ");
    if (find_ctp(root, decl_p) != -1)
      fprintf(header_ptr, "PtrTp");
    else if (find_atp(root, decl_p) != -1)
      fprintf(header_ptr, "ArrTp");
    else
      fprintf(header_ptr, "HybTp");

    if (i >= num_ctp+num_atp)
      fprintf(header_ptr, "H%d", i-num_ctp-num_atp);
    else if (i >= num_ctp)
      fprintf(header_ptr, "A%d", i-num_ctp);
    else
      fprintf(header_ptr, "%d", i);
    fprintf(header_ptr, "%d \t%d\n", j,
            (find_ctp(root, decl_p) != -1 ? find_ctp(root, decl_p) :
             (find_atp(root, decl_p) != -1 ?
              find_atp(root, decl_p)+num_ctp :
              find_htp(root, decl_p)+num_ctp+num_atp )));
  }
}
///////////////////////////////////////////////////////////////////////////
AutoLink_Cfg(root);
///////////////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////////////
// BSize...
///////////////////////////////////////////////////////////////////////////
for (i=0; i<num_ctp+num_atp+num_htp; i++) {
  fprintf(header_ptr, "\n#define BSize");
  if (i<num_ctp)
```

```
        fprintf(header_ptr, "%d  ", i);
      else if (i<num_ctp+num_atp)
        fprintf(header_ptr, "A%d ", i-num_ctp);
      else
        fprintf(header_ptr, "H%d ", i-num_ctp-num_atp);
      fprintf(header_ptr, "\t100");
    }
    for (i=0; i<num_atp+num_htp; i++) {
      fprintf(header_ptr, "\n#define BSize");
      if (i<num_atp)
        fprintf(header_ptr, "A%db", i);
      else
        fprintf(header_ptr, "H%db", i-num_atp);
      fprintf(header_ptr, "\t100");
    }
    fprintf(header_ptr, "\n\nint BSize[CTpUsed+ATpUsed+HTpUsed+1] = { ");
    for (i=0; i<num_ctp+num_atp+num_htp; i++) {
      fprintf(header_ptr, "BSize");
      if (i<num_ctp)
        fprintf(header_ptr, "%d, ", i);
      else if (i<num_ctp+num_atp)
        fprintf(header_ptr, "A%d, ", i-num_ctp);
      else
        fprintf(header_ptr, "H%d, ", i-num_ctp-num_atp);
    }
    fprintf(header_ptr, "0 };\n");

    fprintf(header_ptr, "\nint BSizeb[ATpUsed+HTpUsed+1] = { ");
    for (i=0; i<num_atp+num_htp; i++) {
      fprintf(header_ptr, "BSize");
      if (i<num_atp)
        fprintf(header_ptr, "A%db, ", i);
      else
        fprintf(header_ptr, "H%db, ", i-num_atp);
    }
    fprintf(header_ptr, "0 };\n");

    fprintf(header_ptr, "\n#endif\n");

#endif

////////////////////////////////////////////////////////////////////////////////
// Begin writing "mpitypes.c"
////////////////////////////////////////////////////////////////////////////////

#ifdef STANDALONE

    fprintf(mpitypes_ptr, "#define MaxFields\t50\n\n");

    for (i=0; i<num_ctp; i++) {

      // find the typedef pointer corresponding to the ctp# i
      typedef_p = find_typedef(find_CTp(root, i));

      fprintf(mpitypes_ptr, "MPI_Datatype\t%s;\n",
              find_MPI_Type(root, typedef_p->display_name()));
    }
```

```
#endif


    // First declarations

    fprintf(mpitypes_ptr, "\nvoid Build_MPI_Types()\n{\n");
    for (i=0; i<num_ctp; i++) {

#ifdef STANDALONE
      // find the typedef pointer corresponding to the ctp# i
      typedef_p = find_typedef(find_CTp(root, i));

      fprintf(mpitypes_ptr, "%s\tCType%d;\n", typedef_p->display_name(), i);
#else
      fprintf(mpitypes_ptr, "CTp%d CType%d;\n", i, i);
#endif
    }
#ifndef STANDALONE
    for (i=0; i<num_atp; i++) {
      fprintf(mpitypes_ptr, "ATp%d AType%d;\n", i, i);
    }
    for (i=0; i<num_htp; i++) {
      fprintf(mpitypes_ptr, "HTp%d HType%d;\n", i, i);
    }
#endif

    fprintf(mpitypes_ptr,
            "\nMPI_Aint\tdisp[MaxFields];\nMPI_Datatype\ttype[MaxFields];\n");
    fprintf(mpitypes_ptr,
            "int\t\tblocklen[MaxFields];\nint\t\tbase;\nint\t\ti;\n\n");


    //////////////////////////////////////////////////////////////////////////
    // Check if all MPI_Datatypes are defined
    //////////////////////////////////////////////////////////////////////////

    // First loop: declarations
    for (i=0; i<num_ctp+num_atp+num_htp; i++) {
      // find the typedef pointer corresponding to the XTp #i
      typedef_p = find_typedef(find_XTp(root, i));

      // find the structure which correspond to the XTp #i
      structure = find_struct(root, find_XTp(root, i));

      for (j=0; j<structure->num_fields; j++) {
        if (find_typedef_name(find_decl(structure, j+1)) != NULL
            &&
            find_declarator(structure, j+1)->num_stars == 0)
          if (find_declaration_typedef(root, find_decl(structure, j+1))
              != NULL)
            if ((typedef_p = find_typedef(find_declaration_typedef(root,
                                          find_decl(structure, j+1)))) != NULL)

              if (typedef_p->MPI_Construct == 0)
                Types_declare(root, find_declaration_typedef(root,
                                                   find_decl(structure, j+1)));
      }
```

```
    }

    // Second loop: definitions of additional MPI datatypes

    for (i=0; i<num_ctp+num_atp+num_htp; i++) {
      // find the typedef pointer corresponding to the XTp #i
      typedef_p = find_typedef(find_XTp(root, i));

      // find the structure which correspond to the XTp #i
      structure = find_struct(root, find_XTp(root, i));

      for (j=0; j<structure->num_fields; j++) {
        if (find_typedef_name(find_decl(structure, j+1)) != NULL
            &&
            find_declarator(structure, j+1)->num_stars == 0)
          if (find_declaration_typedef(root, find_decl(structure, j+1))
              != NULL)
            if ((typedef_p = find_typedef(find_declaration_typedef(root,
                                         find_decl(structure,j+1)))) != NULL)

              if (typedef_p->MPI_Construct == 1)
                Types_construct(root, find_declaration_typedef(root,
                                                 find_decl(structure, j+1)));
      }
    }
    ////////////////////////////////////////////////////////////////////////
    // CTp Construction
    ////////////////////////////////////////////////////////////////////////

    Types_construct(root, 1, num_ctp);

#ifndef STANDALONE


    ////////////////////////////////////////////////////////////////////////
    // ATp Construction
    ////////////////////////////////////////////////////////////////////////

    Types_construct(root, 2, num_atp);

    ////////////////////////////////////////////////////////////////////////
    // HTp Construction
    ////////////////////////////////////////////////////////////////////////

    Types_construct(root, 3, num_htp);

#endif

    fprintf(mpitypes_ptr, "}\n");

////////////////////////////////////////////////////////////////////////////
// Begin writing "automap.h"
////////////////////////////////////////////////////////////////////////////

#ifndef STANDALONE

    fprintf(automap_ptr,
```

```
            "/* AutoMap DEFINES */\n\n");

    for (i=0; i<num_ctp; i++) {
      typedef_p = find_typedef(find_CTp(root, i));
      fprintf(automap_ptr, "#define\t%s_AutoNbr\t%d\n",
              typedef_p->display_name(), i);
    }
    for (i=0; i<num_atp; i++) {
      typedef_p = find_typedef(find_ATp(root, i));

      fprintf(automap_ptr,
              "#define\t%s_AutoNbr\t%d\n", typedef_p->display_name(),
              i+num_ctp);
    }
    for (i=0; i<num_htp; i++) {
      typedef_p = find_typedef(find_HTp(root, i));

      fprintf(automap_ptr,
              "#define\t%s_AutoNbr\t%d\n", typedef_p->display_name(),
              i+num_ctp+num_atp);
    }
#endif
  }
  }
#ifndef STANDALONE
  //  close the "userdefs.h" file
  fclose(header_ptr);
#endif

  //  close the "mpitypes.c" file
  fclose(mpitypes_ptr);

#ifndef STANDALONE
  //  close the "automap.h" file

  fclose(automap_ptr);
#endif

  //  close input
  yy_inp2 -> yy_inp_close(yy_true);
  yy_inp1 -> yy_inp_close(yy_true);

  //  delete the objects in reverse order of creation
  delete yy_parser1;
  delete yy_lexer1;
  delete yy_symtab1;
  delete yy_inp1;

  delete yy_parser;
  delete yy_lexer2;
  delete yy_symtab2;
  delete yy_inp2;

  //  call the exit status routine before deleting the error object

  yy_err_exit();
```

```
  delete yy_err2;
  delete yy_err1;

  break;

case 2:  // Fortran-77

  cout << "-- AutoMap for Fortran-77 --"
       << endl;
  break;

case 3:  // Fortran-90

  cout << "-- AutoMap for Fortran-90 --"
       << endl;
  break;

case 4:  // C++

  cout << "-- AutoMap for C++ --"
       << endl;
  break;
}
}
```

Appendix H

prog.c

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "struct.h"

#include "mpitypes.c"

     main(argc,argv)
     int argc;
     char *argv[];
     {
       int my_rank, i;
       int num_ranks;
       MPI_Status status;
       int count;
       int next_rank;
       int tag;

       event_struct ev;
       cmdline      cl;
       Part_struct  ps;
       rect_struct  re;
       cube_struct  cu;
       Treenode     tn;
       DIR          dr;

       MPI_Init(&argc,&argv);

       Build_MPI_Types();

       MPI_Comm_rank(MPI_COMM_WORLD, &my_rank); /* get rank */
       MPI_Comm_size(MPI_COMM_WORLD, &num_ranks); /* get number of ranks */

       ev.pid=15; ev.rate=23.67;

       strcpy(cl.display, "Hello World !");
```

```
        cl.maxiter=26; cl.xmin=-2.5; cl.ymin=-5.0; cl.xmax=12.534;
        cl.ymax=54.67;
        cl.width=10; cl.height=7;

        ps.class=2;
        for(i=0; i<6; i++) ps.d[i]= i+1.2;
        strcpy(ps.b, "abcdef");

        cu.rect1.pt1.x = -4; cu.rect1.pt1.y = -5; cu.rect1.pt2.x = 77;
        cu.rect1.pt2.y = 21; cu.rect2.pt1.x = -344; cu.rect2.pt1.y = -732;
        cu.rect2.pt2.x = 233; cu.rect2.pt2.y = 99;

        tn.Info.Depth = 32; tn.Info.Stop = 4; tn.Info.NodeType = -1;
        tn.Info.funct = 1; tn.Info.TypeData = -1; tn.Info.Index = -87;
        tn.Info.Nchildren = 45; tn.Info.Data = 7.45;

        dr.fd = -356; dr.d.ino = 99999999;
        strcpy(dr.d.name, "AutoMap.yxx");

        count = 1;
        next_rank = 1 - my_rank;

        tag = 0;

        MPI_Send(&ev,count,AutoMap_event_struct,next_rank,tag,
                 MPI_COMM_WORLD);
        MPI_Send(&cl,count,AutoMap_cmdline,next_rank,tag,
                 MPI_COMM_WORLD);
        MPI_Send(&ps,count,AutoMap_Part_struct,next_rank,tag,
                 MPI_COMM_WORLD);
        MPI_Send(&cu,count,AutoMap_cube_struct,next_rank,tag,
                 MPI_COMM_WORLD);
        MPI_Send(&tn,count,AutoMap_Treenode,next_rank,tag,
                 MPI_COMM_WORLD);
        MPI_Send(&dr,count,AutoMap_DIR,next_rank,tag,
                 MPI_COMM_WORLD);

        MPI_Finalize();
}
```

Appendix I

prog2.c

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "struct.h"

#include "mpitypes.c"

event_struct ev;
cmdline      cl;
Part_struct  ps;
cube_struct  cu;
Treenode     tn;
DIR          dr;

     output(file)
     FILE *file;
     {
       int i;

       fprintf(file, "pid val:\t%d\nrate:\t%f\n", ev.pid, ev.rate);

       fprintf(file, "\nmaxiter=%d\nxmin=%f\nymin=%f\nxmax=%f\nymax=%f\n",
               cl.maxiter, cl.xmin, cl.ymin, cl.xmax, cl.ymax);
       fprintf(file, "width=%d\nheight=%d\n",
               cl.width, cl.height);
       fprintf(file, "%s\n", cl.display);

       fprintf(file, "\nclass=%d\n", ps.class);
       for (i=0; i<6; i++) fprintf(file, "d[%d]=%f\n", i, ps.d[i]);
       fprintf(file, "b=%s\n", ps.b);

       fprintf(file, "\nrect1:\n  pt1: %d, %d\n  pt2: %d, %d\n",
               cu.rect1.pt1.x, cu.rect1.pt1.y, cu.rect1.pt2.x, cu.rect1.pt2.y);
       fprintf(file, "rect2:\n  pt1: %d, %d\n  pt2: %d, %d\n",
               cu.rect2.pt1.x, cu.rect2.pt1.y, cu.rect2.pt2.x, cu.rect2.pt2.y);
```

```
  fprintf(file, "\nDepth: %d\nStop: %d\nNodeType: %d\nfunct: %d\n",
          tn.Info.Depth, tn.Info.Stop, tn.Info.NodeType, tn.Info.funct);
  fprintf(file, "TypeData: %d\nIndex: %d\nNchildren: %d\nData: %f\n",
          tn.Info.TypeData, tn.Info.Index, tn.Info.Nchildren, tn.Info.Data
          );
  fprintf(file, "Parent: %d\n", tn.Parent);

  fprintf(file, "\nfd: %d\nino: %d\nname: %s\n",
          dr.fd, dr.d.ino, dr.d.name);
}


main(argc,argv)
int argc;
char *argv[];
{
  int my_rank, i;
  int num_ranks;
  MPI_Status status;
  int count;
  int next_rank;
  int tag;
  FILE *file_ptr, *file_ptr2;


  if ((file_ptr=fopen("/home/fs2a/baland/test/MPI_AutoMap/before","w"))
      == NULL) {
    printf("Can't open file");
    exit(1);
  }

  if ((file_ptr2 =fopen("/home/fs2a/baland/test/MPI_AutoMap/after","w"))
      == NULL) {
    printf("Can't open file2");
    exit(1);
  }

  MPI_Init(&argc,&argv);

  Build_MPI_Types();

  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank); /* get rank */
  MPI_Comm_size(MPI_COMM_WORLD, &num_ranks); /* get number of ranks */

  ev.pid=0; ev.rate=0.0;

  strcpy(cl.display, "Quick lunch ?");
  cl.maxiter=0; cl.xmin=0.0; cl.ymin=0.0; cl.xmax=0.0; cl.ymax=0.0;
  cl.width=0; cl.height=0;

  ps.class=0;
  for(i=0; i<6; i++) ps.d[i]=0.0;
  strcpy(ps.b, "aaaaaa");

  cu.rect1.pt1.x = 0; cu.rect1.pt1.y = 0; cu.rect1.pt2.x = 0;
  cu.rect1.pt2.y = 0; cu.rect2.pt1.x = 0; cu.rect2.pt1.y = 0;
  cu.rect2.pt2.x = 0; cu.rect2.pt2.y = 0;
```

```
  tn.Info.Depth = 0; tn.Info.Stop = 0; tn.Info.NodeType = 0;
  tn.Info.funct = 0; tn.Info.TypeData = 0; tn.Info.Index = 0;
  tn.Info.Nchildren = 0; tn.Info.Data = 0.0;

  dr.fd = 0; dr.d.ino = 0;
  strcpy(dr.d.name, "struct.h");

  count = 1;
  next_rank = 1-my_rank;
  tag = 0;

  output(file_ptr);

  MPI_Recv(&ev,count,AutoMap_event_struct,next_rank,
           MPI_ANY_TAG,MPI_COMM_WORLD,&status);
  MPI_Recv(&cl,count,AutoMap_cmdline,next_rank,
           MPI_ANY_TAG,MPI_COMM_WORLD,&status);
  MPI_Recv(&ps,count,AutoMap_Part_struct,next_rank,
           MPI_ANY_TAG,MPI_COMM_WORLD,&status);
  MPI_Recv(&cu,count,AutoMap_cube_struct,next_rank,
           MPI_ANY_TAG,MPI_COMM_WORLD,&status);
  MPI_Recv(&tn,count,AutoMap_Treenode,next_rank,
           MPI_ANY_TAG,MPI_COMM_WORLD,&status);
  MPI_Recv(&dr,count,AutoMap_DIR,next_rank,
           MPI_ANY_TAG,MPI_COMM_WORLD,&status);

  output(file_ptr2);

  MPI_Finalize();
  fclose(file_ptr);
  fclose(file_ptr2);
}
```

Appendix J

struct.h

```
/** Test file for AutoMap, stand-alone version **/

/*~ AutoMap_Begin */

typedef struct {
  int pid;                /* -1 indicates null */
  int row_src, col_src;  /* range [1..n], [1..m] */
  int row_dst, col_dst;  /* range [0..n], [0..m+1] */
  float rate;
  float priority;
} event_struct /*~ AutoMap_CTpUsed */;

/*****************************************************************************/

struct commandline {
  char     display[50];
  int      maxiter;
  double   xmin, ymin;
  double   xmax, ymax;
  int      width;
  int      height;
};

typedef struct commandline cmdline /*~ AutoMap_CTpUsed */;

/*****************************************************************************/

struct Partstruct
{
  int class;
  double d[6];
  char b[7];
};

typedef struct Partstruct Part_struct /*~ AutoMap_CTpUsed */  ;

/*****************************************************************************/
```

```
typedef struct point {
  int x;
  int y;
} pt_struct;

struct rect {
  pt_struct pt1;
  pt_struct pt2;
};

typedef struct rect rect_struct;

struct cube {
  rect_struct rect1;
  rect_struct rect2;
};

typedef struct cube cube_struct /*~ AutoMap_CTpUsed */ ;

/*****************************************************************************/

struct Treeinfo {
  int Depth;
  int Stop;
  int NodeType;   /* 0 means data; 1 means a function; -1 means not assigned */
  int funct;      /* integer indexes into the function table                 */
  int TypeData;   /* 0 means BiggerReal; 1 means array data; -1 means not
                     assigned                                                */
  int Index;      /* indexes into array data                                 */
  int Nchildren;
  float Data;     /* for putting BiggerReals into   */
};


typedef struct Treeinfo TreeInfo;
typedef struct tnode  *Treeptr;

struct tnode {
         TreeInfo   Info;
         Treeptr    Parent; /* pointer to node's parent */
         Treeptr    left;
         Treeptr    middle;
         Treeptr    right;
         Treeptr    FarRight;
                };

typedef struct tnode Treenode /*~ AutoMap_CTpUsed */ ;

/*****************************************************************************/
/* C book p.180 */

typedef struct {
  long ino;
  char name[15];
} Dirent;

struct directory {
```

```
  int fd;
  Dirent d;
};

typedef struct directory DIR /*~ AutoMap_CTpUsed */ ;

/*~ AutoMap_End */
```

Appendix K

mpitypes.c

```
#define MaxFields       50

MPI_Datatype    AutoMap_event_struct;
MPI_Datatype    AutoMap_cmdline;
MPI_Datatype    AutoMap_Part_struct;
MPI_Datatype    AutoMap_cube_struct;
MPI_Datatype    AutoMap_Treenode;
MPI_Datatype    AutoMap_DIR;

void Build_MPI_Types()
{
event_struct    CType0;
cmdline CType1;
Part_struct     CType2;
cube_struct     CType3;
Treenode        CType4;
DIR     CType5;

MPI_Aint        disp[MaxFields];
MPI_Datatype    type[MaxFields];
int             blocklen[MaxFields];
int             base;
int             i;

pt_struct       TpDef0;
MPI_Datatype    AutoMap_pt_struct;
rect_struct     TpDef1;
MPI_Datatype    AutoMap_rect_struct;
TreeInfo        TpDef2;
MPI_Datatype    AutoMap_TreeInfo;
Dirent  TpDef3;
MPI_Datatype    AutoMap_Dirent;

/* pt_struct */
MPI_Address(&TpDef0.x,  &disp[0]);
MPI_Address(&TpDef0.y,  &disp[1]);
type[0] = MPI_INT;
type[1] = MPI_INT;
```

```
blocklen[0] = 1;
blocklen[1] = 1;
base = disp[0]; for(i=0; i<2; i++) disp[i]-=base;
MPI_Type_struct(2, blocklen, disp, type, &AutoMap_pt_struct);
MPI_Type_commit(&AutoMap_pt_struct);

/* rect_struct */
MPI_Address(&TpDef1.pt1,        &disp[0]);
MPI_Address(&TpDef1.pt2,        &disp[1]);
type[0] = AutoMap_pt_struct;
type[1] = AutoMap_pt_struct;
blocklen[0] = 1;
blocklen[1] = 1;
base = disp[0]; for(i=0; i<2; i++) disp[i]-=base;
MPI_Type_struct(2, blocklen, disp, type, &AutoMap_rect_struct);
MPI_Type_commit(&AutoMap_rect_struct);

/* TreeInfo */
MPI_Address(&TpDef2.Depth,      &disp[0]);
MPI_Address(&TpDef2.Stop,       &disp[1]);
MPI_Address(&TpDef2.NodeType,   &disp[2]);
MPI_Address(&TpDef2.funct,      &disp[3]);
MPI_Address(&TpDef2.TypeData,   &disp[4]);
MPI_Address(&TpDef2.Index,      &disp[5]);
MPI_Address(&TpDef2.Nchildren,  &disp[6]);
MPI_Address(&TpDef2.Data,       &disp[7]);
type[0] = MPI_INT;
type[1] = MPI_INT;
type[2] = MPI_INT;
type[3] = MPI_INT;
type[4] = MPI_INT;
type[5] = MPI_INT;
type[6] = MPI_INT;
type[7] = MPI_FLOAT;
blocklen[0] = 1;
blocklen[1] = 1;
blocklen[2] = 1;
blocklen[3] = 1;
blocklen[4] = 1;
blocklen[5] = 1;
blocklen[6] = 1;
blocklen[7] = 1;
base = disp[0]; for(i=0; i<8; i++) disp[i]-=base;
MPI_Type_struct(8, blocklen, disp, type, &AutoMap_TreeInfo);
MPI_Type_commit(&AutoMap_TreeInfo);

/* Dirent */
MPI_Address(&TpDef3.ino,        &disp[0]);
MPI_Address(TpDef3.name,        &disp[1]);
type[0] = MPI_LONG;
type[1] = MPI_CHAR;
blocklen[0] = 1;
blocklen[1] = 1*(15);
base = disp[0]; for(i=0; i<2; i++) disp[i]-=base;
MPI_Type_struct(2, blocklen, disp, type, &AutoMap_Dirent);
MPI_Type_commit(&AutoMap_Dirent);
```

```
/* event_struct */
MPI_Address(&CType0.pid,        &disp[0]);
MPI_Address(&CType0.row_src,    &disp[1]);
MPI_Address(&CType0.col_src,    &disp[2]);
MPI_Address(&CType0.row_dst,    &disp[3]);
MPI_Address(&CType0.col_dst,    &disp[4]);
MPI_Address(&CType0.rate,       &disp[5]);
MPI_Address(&CType0.priority,   &disp[6]);
type[0] = MPI_INT;
type[1] = MPI_INT;
type[2] = MPI_INT;
type[3] = MPI_INT;
type[4] = MPI_INT;
type[5] = MPI_FLOAT;
type[6] = MPI_FLOAT;
blocklen[0] = 1;
blocklen[1] = 1;
blocklen[2] = 1;
blocklen[3] = 1;
blocklen[4] = 1;
blocklen[5] = 1;
blocklen[6] = 1;
base = disp[0]; for(i=0; i<7; i++) disp[i]-=base;
MPI_Type_struct(7, blocklen, disp, type, &AutoMap_event_struct);
MPI_Type_commit(&AutoMap_event_struct);

/* cmdline */
MPI_Address(CType1.display,     &disp[0]);
MPI_Address(&CType1.maxiter,    &disp[1]);
MPI_Address(&CType1.xmin,       &disp[2]);
MPI_Address(&CType1.ymin,       &disp[3]);
MPI_Address(&CType1.xmax,       &disp[4]);
MPI_Address(&CType1.ymax,       &disp[5]);
MPI_Address(&CType1.width,      &disp[6]);
MPI_Address(&CType1.height,     &disp[7]);
type[0] = MPI_CHAR;
type[1] = MPI_INT;
type[2] = MPI_DOUBLE;
type[3] = MPI_DOUBLE;
type[4] = MPI_DOUBLE;
type[5] = MPI_DOUBLE;
type[6] = MPI_INT;
type[7] = MPI_INT;
blocklen[0] = 1*(50);
blocklen[1] = 1;
blocklen[2] = 1;
blocklen[3] = 1;
blocklen[4] = 1;
blocklen[5] = 1;
blocklen[6] = 1;
blocklen[7] = 1;
base = disp[0]; for(i=0; i<8; i++) disp[i]-=base;
MPI_Type_struct(8, blocklen, disp, type, &AutoMap_cmdline);
MPI_Type_commit(&AutoMap_cmdline);

/* Part_struct */
MPI_Address(&CType2.class,      &disp[0]);
```

```
MPI_Address(CType2.d,   &disp[1]);
MPI_Address(CType2.b,   &disp[2]);
type[0] = MPI_INT;
type[1] = MPI_DOUBLE;
type[2] = MPI_CHAR;
blocklen[0] = 1;
blocklen[1] = 1*(6);
blocklen[2] = 1*(7);
base = disp[0]; for(i=0; i<3; i++) disp[i]-=base;
MPI_Type_struct(3, blocklen, disp, type, &AutoMap_Part_struct);
MPI_Type_commit(&AutoMap_Part_struct);

/* cube_struct */
MPI_Address(&CType3.rect1,      &disp[0]);
MPI_Address(&CType3.rect2,      &disp[1]);
type[0] = AutoMap_rect_struct;
type[1] = AutoMap_rect_struct;
blocklen[0] = 1;
blocklen[1] = 1;
base = disp[0]; for(i=0; i<2; i++) disp[i]-=base;
MPI_Type_struct(2, blocklen, disp, type, &AutoMap_cube_struct);
MPI_Type_commit(&AutoMap_cube_struct);

/* Treenode */
MPI_Address(&CType4.Info,       &disp[0]);
MPI_Address(&CType4.Parent,     &disp[1]);
MPI_Address(&CType4.left,       &disp[2]);
MPI_Address(&CType4.middle,     &disp[3]);
MPI_Address(&CType4.right,      &disp[4]);
MPI_Address(&CType4.FarRight,   &disp[5]);
type[0] = AutoMap_TreeInfo;
type[1] = MPI_INT;
type[2] = MPI_INT;
type[3] = MPI_INT;
type[4] = MPI_INT;
type[5] = MPI_INT;
blocklen[0] = 1;
blocklen[1] = 1;
blocklen[2] = 1;
blocklen[3] = 1;
blocklen[4] = 1;
blocklen[5] = 1;
base = disp[0]; for(i=0; i<6; i++) disp[i]-=base;
MPI_Type_struct(6, blocklen, disp, type, &AutoMap_Treenode);
MPI_Type_commit(&AutoMap_Treenode);

/* DIR */
MPI_Address(&CType5.fd, &disp[0]);
MPI_Address(&CType5.d,  &disp[1]);
type[0] = MPI_INT;
type[1] = AutoMap_Dirent;
blocklen[0] = 1;
blocklen[1] = 1;
base = disp[0]; for(i=0; i<2; i++) disp[i]-=base;
MPI_Type_struct(2, blocklen, disp, type, &AutoMap_DIR);
MPI_Type_commit(&AutoMap_DIR);
}
```

Bibliography

[1] MPI: A Message Passing Interface Standard. HTML document, 1994. http://www.mcs.anl.gov/Projects/mpi/index.html.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley, 1988.

[3] D. Burns and Raja B. Daoud. Lam: An open cluster environment for mpi. In *Supercomputing Symposium '94*, June 1994. Toronto, Canada, code available at http://www.osc.edu/lam.html.

[4] Compiler Resources, Hopkinton, MA. *Yacc++ and the Language Objects Library Reference Guide*, 1996. email address compress@world.std.com.

[5] Judith E. Devaney. MPI/LAM course. HTML document. http://www.itl.nist.gov/div895/sasg/parallel/mpi/lam/mpilamcourse.html.

[6] Judith Ellen Devaney, Martial Michel, Jasper Peeters, and Koen Vrielink. AutoLink: An MPI C Library For Sending and Receiving Dynamic Data Structures. Technical report, NIST, April 1997. http://www.itl.nist.gov/div895/sasg/parallel/.

[7] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, Cambridge, MA, 1994.

[8] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language, second edition*. Prentice Hall PTR, Englewood Cliffs, NJ, 1988.

[9] K. H. J. Vrielink, E. C. Baland, and J. E. Devaney. AutoLink: An MPI Library for Sending and Receiving Dynamic Data Structures. In *International Conference on Parallel Computing*. University of Minnesota Supercomputer Institute, october 3-4, 1996.