**Imperial College of Science,
Technology and Medicine**

**(University of London)**

**Department of Computing**

# *A C to C++ Converter*

**by**

***Saigol, Z.A.***

**Submitted in partial fulfilment of the
requirements for the MSc Degree in
Engineering of the University of
London and for the Diploma of
Imperial College of Science,
Technology and Medicine.**

**September 1996**

## Abstract

This report describes the development of a tool for converting C programs into C++.

Structs in the original C program become classes in the C++ program, and functions from the original program are turned into members of the new class. The user specifies which structs and functions to convert, and the tool performs the required syntactic modifications. Struct declarations are replaced by class declarations and function declarations are replaced by member function declarations. Inside member functions, the original struct variable is removed from accesses to data members. Also function calls are replaced by member function calls where one of the parameters has been turned into the receiver; in the special case where the function call appears inside another member function, the receiver is omitted.

A promising new approach, pattern matching against the abstract syntax tree of the program to be converted, was developed to describe and perform these conversions. This approach may have applications in many other areas concerned with program transformations.

## Acknowledgements

I would like to thank firstly my supervisor, Dr Sophia Drossopoulou, for being extremely helpful when I was stuck on technical points, for her (mostly successful) attempts to steer me away from the odd misguided approach, and for proof reading this report.

I would also like to acknowledge Ian Moor, for some invaluable assistance with an unruly computer program.

**ABSTRACT** ..... **ii**

**ACKNOWLEDGEMENTS** ..... **iii**

**1. INTRODUCTION** ..... **1**

1.1. AIMS ..... 1
1.2. ACHIEVEMENTS ..... 1
1.3. GUIDE TO THIS REPORT ..... 2

**2. BACKGROUND** ..... **4**

2.1. PROBLEM OF LEGACY SOFTWARE ..... 4
2.2. OBJECT ORIENTATION - A SOLUTION? ..... 4
2.3. TRANSFORMATIONS ..... 5
2.4. SCOPE OF THIS PROJECT ..... 6

**3. ANALYSIS** ..... **8**

3.1. APPROACH ..... 8
3.2. OVERVIEW ..... 8
3.3. ISSUES RELATING TO TYPE I ..... 11
*3.3.1 Non - & arguments* ..... *11*
*3.3.2. CTB return type* ..... *12*
*3.3.3. Protect Specification* ..... *15*
*3.3.4. Typedefs* ..... *15*
*3.3.5. Multiple CTB arguments* ..... *16*
*3.3.6. Calling MFTBs* ..... *17*
3.4. ISSUES RELATING TO TYPE II ..... 18
*3.4.1. Introduction* ..... *18*
*3.4.2. Conversions* ..... *19*
*3.4.3. Problems* ..... *20*
3.5. ISSUES RELATING TO TYPE III ..... 21
3.6. REQUIREMENTS FOR SEMANTIC PROCESSING ..... 23
3.7. SUMMARY OF PROGRAM REQUIREMENTS ..... 25

**4. DESIGN** ..... **26**

4.1. SYNTAX TREE ..... 26
*4.1.1. Background* ..... *26*
*4.1.2. Tools* ..... *28*
*4.1.3. Structure of ctrans* ..... *29*
4.2. CHOICE OF CPPP AS FRONT END PARSER ..... 30
4.3. DESIGN OF THE CONVERSION ENGINE ..... 31
*4.3.1. Approach* ..... *31*
*4.3.2. Alternatives: intelligent tree or pattern matching* ..... *32*
4.4. CHOICE OF PATTERN MATCHING ..... 35

**5. PATTERNS** ..... **37**

5.1. BASIC IDEAS ..... 37
*5.1.1. Concepts* ..... *37*
*5.1.2. Stars* ..... *37*
*5.1.3. Function headers* ..... *38*

*5.1.4. Macros* ..... *38*
*5.1.5. Sub - patterns* ..... *38*
*5.1.6. Summary and example* ..... *40*
5.2. EXTENSIONS ..... 46
*5.2.1. Number to find* ..... *46*
*5.2.2. Comparisons with nodes* ..... *47*
*5.2.3. Remove son and concat* ..... *47*
*5.2.4. Special creates* ..... *49*
*5.2.5. Names of nodes* ..... *50*
*5.2.6. Star nodes with sons* ..... *50*
5.3. COMPLETE DESCRIPTION OF PATTERN LANGUAGE ..... 51
*5.3.1. Application* ..... *51*
*5.3.2. Comparison with other pattern matching algorithms* ..... *51*
*5.3.3. Node components* ..... *52*
*5.3.4. Conventions for descriptions* ..... *52*
*5.3.5. If Find nodes* ..... *54*
*5.3.6. Replace nodes* ..... *54*
5.4. ACTUAL PATTERNS ..... 55
*5.4.1. Creating* ..... *55*
*5.4.2. List of basic patterns* ..... *56*
*5.4.3. Full example* ..... *56*

**6. IMPLEMENTATION ..... 58**

6.1. FUNCTIONALITY ..... 58
6.2. CTRANS STRUCTURE ..... 58
6.3. CPPP ..... 59
*6.3.1. Compilation* ..... *59*
*6.3.2. Structure* ..... *60*
*6.3.3. Printing the tree as code* ..... *62*
6.4. APPLYING PATTERNS ..... 63
*6.4.1. Initial object and method ideas* ..... *63*
*6.4.2. Modify or Duplicate?* ..... *66*
*6.4.3. Cppp Syntax Trees* ..... *68*
*6.4.4. Final object design* ..... *69*
*6.4.5. Final application algorithm* ..... *71*
6.5. MAINTENANCE INFORMATION AND UTILITY CODE ..... 74
*6.5.1. All files* ..... *74*
*6.5.2. Storage of patterns* ..... *75*
*6.5.3. Lists and star-lists* ..... *75*
*6.5.4. ct_local* ..... *76*
*6.5.5. ctwords* ..... *76*
*6.5.6. cttemplates* ..... *77*
*6.5.7. Conventions* ..... *77*
6.6. TREE MANAGER ..... 77
*6.6.1. Introduction* ..... *77*
*6.6.2. Finding information for the user interface* ..... *78*
*6.6.3. Finding semantic information* ..... *79*
*6.6.4. Special matches* ..... *80*
*6.6.5. Special create - CTBName* ..... *81*

*6.6.6. Special create - MFTB Headers* ..... *81*
*6.6.7. Summary of tree and Cppp use* ..... *82*
6.7. RE-SCANNING ..... 84
*6.7.1. Checking against other nodes* ..... *84*
*6.7.2. ReScan* ..... *85*
*6.7.3. Extent of star-number visibility* ..... *89*
6.8. REMOVE-SON AND CONCAT ..... 90
6.9. INTERFACE ..... 92
6.10. PATTERN TRANSLATOR ..... 94
6.11. INTEGRATION AND TESTING ..... 96
6.12. RESULTS ..... 97
*6.12.1 Array-based queue* ..... *97*
*6.12.2. Dynamic stack* ..... *99*
*6.12.3. Vectors* ..... *102*
*6.12.4. List* ..... *104*

**7. CONCLUSIONS ..... 107**

7.1. GENERAL ..... 107
7.2. COMPLEXITY OF C ..... 109
7.3. PATTERN MATCHING ..... 110
*7.3.1. Overall evaluation* ..... *110*
*7.3.2. Pattern language syntax* ..... *112*
*7.3.3. Technical drawbacks* ..... *113*
*7.3.4. Summary* ..... *114*
7.4. PROJECT EVALUATION ..... 115
*7.4.1. Decisions* ..... *115*
*7.4.2. Time management* ..... *116*
7.5. FUTURE WORK ..... 118
*7.5.1. Updates for patterns* ..... *118*
*7.5.2. Updates for program* ..... *118*
*7.5.3. Additional patterns* ..... *119*
*7.5.4. Types II and III* ..... *120*
*7.5.5. Extensions to programs* ..... *121*
*7.5.6. Advanced extensions* ..... *123*
7.6 SUMMARY ..... 124

**REFERENCES ..... 126**

**APPENDIX A ..... 128**

A1 TYPEDEF-STRUCT ..... 128
A2 MFTB-ARG-PTR1 ..... 130

**APPENDIX B ..... 133**

<STD-FUNC> ..... 133
B1 TYPEDEF-STRUCT ..... 133
B2 STRUCT-ONLY ..... 134
B3 TYPEDEF-FOR-CTB ..... 135
B4 TYPEDEF-FOR-PTR-CTB ..... 135
B5 NON-MFTB ..... 135

B6 MFTB-ARG-ADDR ....135
B7A MFTB-ARG-PTR1 ....136
B7B MFTB-ARG-PTR2 ....136
B8 MFTB-ARG-VALUE ....137
B9A MFTB-RETURN-PTR1 ....138
B9B MFTB-RETURN-PTR2 ....139
B10 MFTB-RETURN-VALUE ....139

**APPENDIX C ....141**

CPPPAST OBJECT HEIRARCHY ....141

**CTRANS USERS' GUIDE ....145**

INTRODUCTION ....145
SAMPLE SESSION ....145
CONVERSIONS PERFORMED ....150
CONVERSIONS NOT PERFORMED ....150
POSSIBLE BUGS IN OUTPUT ....150

**PATTERN WRITERS' GUIDE ....152**

PATTERN WRITING ....152
*Overview ....152*
*If-find node components ....152*
*Replace-with node components ....154*
*Trans-spec components ....154*
*If-replace-spec components ....154*
*Match and replace specifications ....155*
*Notes ....155*
PT ....156
CTP SYNTAX ....156
ADDING PATTERNS TO CTRANS ....159

# 1. INTRODUCTION

## *1.1. Aims*

Object-orientation is a new programming paradigm that has many advantages over the conventional (imperative) approach. As well as being easier to modify, object oriented (OO) code is easily re-used in a different application.

However, the vast majority of programs in use today have been developed using the imperative paradigm, and often significant benefits can be gained from re-writing this code in an OO language. The aim of this project was to develop an application to assist with this, with C++ being the target language and C the input language.

The essence of transformations to be performed by the application is to convert a C struct into a C++ class, and make functions in the original program into member functions of the new class in the converted program. Both the struct and the functions to become members are to be selected by the user. This contrasts with previous work on creating OO programs from imperative code, which has concentrated on algorithms to automatically detect which structures in the original program should become objects; as a result, the applications developed have only progressed to a prototype stage. Also, converting C to C++ has not previously been attempted.

## *1.2. Achievements*

Despite encountering several setbacks (which are detailed in sections 6.3.1 and 6.4.3) in the design and implementation of the program, a powerful application was developed and tested successfully on several C programs.

Ctrans (the name given to the application, for C transformer) uses a ‘pattern matching’ algorithm, where the transformations to be performed on the input program are represented in the form of a pattern. These patterns are very easy to modify in order to describe different conversions, or to add to so that new conversions can be performed. The patterns consist of a sub-pattern of syntax tree nodes to look for in the syntax tree of the input program, and another sub-pattern to replace this with if any examples of it are found.

Figure 1.1 shows the structure of ctrans. Cppp is the parsing/semantic analysis package used (described in section 6.3), and this creates and manages the syntax tree for the input program. However, a module to output this tree (as C++) had to be written. Ctrans, the user interface, and a program to translate patterns into C++ files were also written.

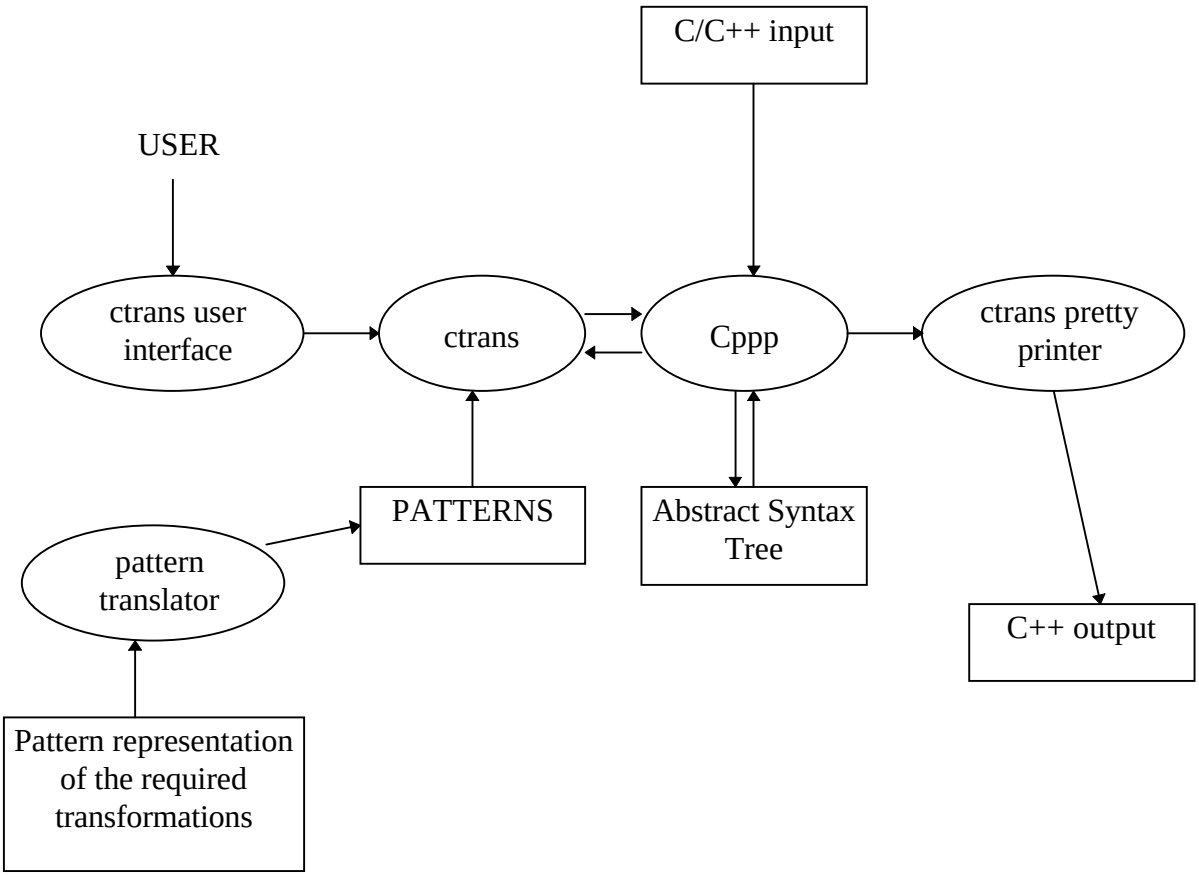



**Figure 1.1. Overall structure of ctrans**

Ctrans itself can be viewed as just a sophisticated package to apply a pattern, with some additional I/O facilities, and the functionality is really provided by the patterns. To write these, sample C programs were found and a considerable amount of analysis was performed on these. This enabled the syntactic changes the that would need to be made to produce a reasonable OO output to be determined.

### *1.3. Guide to this report*

Section 2 covers the background to the project, including why a tool such as ctrans should be desired, and the work of other people in addressing the similar problems.

Section 3 documents the findings on conversions that the tool will be required to perform. The transformations actually done by ctrans are listed in section 3.3, and section 3.6 describes the semantic information that has to be found about the input program.

Section 4 is a brief guide to the high-level design decisions taken, and also contains a summary of the packages looked at to assist with creating a useful representation of the input program. The choice of the pattern matching algorithm, and the alternative options, are covered in sections 4.3 and 4.4.

Section 5 contains a description of the evolution and semantics of the pattern 'language' that was developed to describe program transformations, and forms a key part of the project.

Section 6 is the main implementation section, which deals firstly with the format and semantics of the patterns applied by the tool to the input code. It introduces the program structure and lower level design decisions, and then has a guide to the code of the tool, including the file structure and data used by the program. The ancillary program developed to translate patterns into a form understood by the program is described, and finally the output and efficiency of the tool is documented.

Section 7 is the conclusions section, where the overall concept, tool itself and pattern matching algorithm are evaluated. There are also subsections describing the considerable possibilities for future work in this area.

# 2. BACKGROUND

## *2.1. Problem of legacy software*

Many large organisations have software systems that were originally designed in the sixties or seventies, which have had considerable resources invested in their development. Unfortunately such applications, called legacy software, are often badly structured and very difficult to maintain, and can cost the company a great deal of money. For example, Lano and Haughton [1] performed a summary of recent research, and found that the cost of maintaining software often amounts to 80 per cent of the total cost of the software.

Software can become a legacy for several reasons. Firstly, as outlined by Jacobson and Lindstrom [2], changes made to the software over the years destroy any structure it originally had. This makes further changes more time consuming and expensive, and finally it is not economically viable to make a required change.

Another reason could be that the company needs to upgrade to a more modern architecture or hardware set up, for example a client- server system, and substantial changes to the current system are needed for it to work in the new environment.

Gall *et al*. [3] describe the situation thus:

"Legacy systems are an increasing problem for IT groups in large organisations. Such systems contain a large amount of information and knowledge about a certain application domain, but they are usually under-documented, written in a dated language and the original design has been blurred as a result of several semistructured or unstructured maintenance operations. Additionally, many of these systems do not meet the growing demands of the business, and are costly to operate."

## *2.2. Object Orientation - a solution?*

There have been considerable advances in the approaches used to engineer large programs in the last three decades, and it is far less likely that a modern software application is going to become a legacy. New paradigms have been developed, such as high level languages, procedural abstraction, top down design, and abstract data types (ADTs), and these have made software easier to maintain.

A structured approach to the development of software has also become the norm, and it has been found that code produced in accordance with a detailed design, and in a structured manner, is much more easily modified and maintained.

One of the most promising recent developments has been the object oriented approach, which according to Gall *et al* [3] "is one of the well- known principles to realise "design for change"". Indeed, as well as the benefits in the code produced, Winston [4] notes that "The productivity of C++ programmers generally exceeds the productivity of C programmers".

The key feature of object oriented languages is the class system. A class is similar to an ADT in that it encapsulates data and procedures for modifying that data, but classes build on ADTs by allowing features such as inheritance, polymorphism and dynamic binding. These facilities make the re-use of classes much easier, and re-using software helps overcome many of the difficulties encountered with software development and maintenance. If existing code is used, it will have been tested thoroughly, and if new code is written which is going to be re-used, more time can be spent on making it easily understood and changed.

Another benefit of object orientation comes from basing the structure of the software on the application domain, rather than on the algorithms that will be used to create the solution. This helps because the domain of a problem is often stable, whereas the requirements for software functionality usually change.

### *2.3. Transformations*

Given the advantages of object orientation, it would present an obvious solution to some of the problems of legacy software if it could be re- engineered with an object oriented structure. In such a transformation, the syntax and structure of the code would change, but it would generally be possible to keep the original procedures, performing the same tasks. This would mean that all the valuable and encoded domain knowledge of the old application could be kept in the new, re-structured package.

Unfortunately, converting a large program to be object oriented is not easy - the idea of basing the variables in a program on the real- world domain needs to be used in a design from the start. Also, an object oriented approach can imply a different view of how parts of the program communicate, which could require extensive modifications.

However, transformations of programs to a new, but not object oriented, structure are very common. Onoma *et al* [5] describe examples of several such transformations, as well as a highly sophisticated tool (RE-ENGINEERING) to help with such transformations, and a paper by Choi and Scacchi [6] is one example of many that describe re-structuring processes.

Furthermore, on the topic of reverse engineering software, Lano and Haughton [1] note that: "More recently, .... a realisation of the centrality of maintenance in the production and continued utility of software systems has become more widespread."

This growing interest has also extended to research into the task of producing object oriented systems from old procedural software. Examples of research found into this are described below.

Jacobson and Lindstrom [2] have taken a practical approach, based on the belief that in terms of risk and resources, it is impossible to convert a whole system at once, and so ways of integrating OO and non - OO parts are needed. The goal of their system is to identify data structures that would make good objects in the transformed version. However they also mostly eschew tools, which are the key interest for this project, believing that the methodology used for reengineering the software system is of prime importance.

Gall *et al* [3] have designed a process involving human experts and tools to produce a complete OO translation of the original system. They believe that domain knowledge is essential to produce a reasonable OO design, and this is provided by a human expert who creates a forward engineered design. This design is then reconciled, again with human intervention, with a design reverse engineered from the existing code. The resulting object design is given to a source code adapter, which converts Pascal syntax into object Pascal.

Finally, Ong and Tsai [7] have performed particularly relevant work on converting FORTRAN 77 into C++. Their tool firstly parses the input program and 'decouples' it from F77 - by removing FORTRAN specific constructs. A data flow analysis phase subsequently identifies potential objects and classes, and then a target code generator creates the C++ code from the representation of the modified program. However, the details of the conversion algorithm are not given.

### *2.4. Scope of this project*

The project differs from the previous work described above in two key ways: firstly, it converts C code into C++, and secondly, the aim is to provide a useful tool for user guided code translation, rather than to create a structured system for extracting objects from imperative programs. The language level conversions, which consist of transforming aggregate data types into classes and converting functions into members for this class, are however similar to [3] and [7]. In [3], the conversion is from Pascal to Object Pascal, which is considerably more straightforward; however, [7] convert f77 to C++, but do not give any details of the methods used for transforming syntax.

Writing in 1993, Ong and Tsai [7] said: "some translators are available to literally translate old code into a new object-oriented language. .... Unfortunately, such translators do not recognise or create objects and therefore do not create object oriented code."

The conversion of C into C++ with objects is felt to be particularly useful, due mostly to the large amount of commercial software that is written in C. Also, C++ is the most commonly used object oriented language, with the result

that there is a lot of support for it. It is similar to C, which is familiar to a lot of programmers, and it is a fast and flexible language.

The aims in creating a language-level program conversion tool were twofold; firstly to demonstrate the benefits of this to the maintainability of large software such as legacy systems, and secondly to provide a tool that will be useful to anyone faced with the task of adding OO features to a C program.

If effectiveness of the conversion tool could be shown adequately, it could inspire a more complete version or development. For re- engineering, this would probably have to be used in the context of a systematic methodology for finding information about the existing system, and deciding what to make into classes and objects. Even so it would still relieve the maintainers of the long and tedious task of changing the syntax of the program, and hopefully provide a bit of extra intelligence also. The research prototypes mentioned above are further away from being usable, and would still need human guidance.

It is also hoped that the tool could have a use at a lower level, where a programmer has quickly created a program in a conventional, imperative manner, and then realised that it will need to be used and modified more extensively than originally thought. If the programmer has written the program in a fairly logical way, it may well be obvious to him/ her what the objects should be, and then the tool will be able to quickly produce a much neater version of this program.

The value of this can be seen from the experiences of Fitzpatrick [8], who manually converted a C program for testing graphics adapters into C++. He found the OO version produced a smaller executable, and concluded that "many C programs can benefit from object- oriented extensions, even when a complete rewrite is impractical."

# 3. ANALYSIS

## *3.1. Approach*

The first stage of the project was to find the kind of syntactic modifications that ctrans would be required to make, and this was done by performing manual conversions on a range of example programs. Knowledge of good object oriented style was used to select programs that would make good examples, and also in deciding how to modify these examples. The modifications made were then abstracted into code templates or patterns, representing the changes needed in different situations.

Finding appropriate example programs proved to be surprisingly difficult, given the high usage of C. One source of programs initially investigated was C programs written to perform a useful task, which were available plentifully via ftp. However such programs, for example Unix utilities, proved to be far too long and generally too unstructured to make good, clear test programs. Another source was the many C tutorials available on the World Wide Web, which often had example programs included with them. Unfortunately these programs proved too short and simple to be used as examples, usually not containing any C structs at all, and certainly none that would make good classes.

A source that produced some usable, but not ideal, examples was programs written by the author for student exercises, both in C and in Pascal. The Pascal programs were converted to C, but were atypical of the style used by C programmers - notably in that they made no use of pointers.

The best source proved to be C textbooks. The example programs found in C++ textbooks were also looked at, but converting them to C and then back to C++ seemed very artificial and not likely to represent the transformations that would be required of the tool. The C textbooks proved more useful, with examples being taken from Hanly *et al* [9], Roberts [10], and Kruse *et at* [11], but still only one or two reasonable programs were found in each book.

In fact, many of the conversions were only inspired by the example programs, as it was often possible to see that if the example was slightly modified or extended, more changes to, and/or conditions on, the code would be required.

## *3.2. Overview*

The fundamental conversion performed on the example programs was to create a C++ class that did not exist in the original program, and to make functions that did exist in the input program into members of this class. However, it became apparent that there are three distinct ways of forming the new class, depending on the format of the data used in the original program. The first, more

straightforward case was when a C struct exists in the non- OO program, and this is converted into a class in the transformed program. Also certain procedures, probably with a variable of this struct type as an argument, become member functions of the new class. From now on, this kind of conversion will be referred to as Type I.

The second case is where no struct exists in the original program, but procedures do exist which manipulate a consistent set of variables. An example could be a set of functions which take two integer arguments representing a fraction. Here a new class should be created, and appropriate objects of this class declared and used in the transformed program. This is a Type II transformation.

The final Type III transformations convert a set of global variables into a class. For these conversions, the new member functions will not originally have had any data members of the new class as arguments, as these data members were declared globally.

It will be useful at this point to introduce some terminology for describing transformations. For a Type I conversion, the struct that is to be turned into a class is known as the Class-To-Be (CTB), and this term will also be used generally for the class that is to be created. Also, any function that is to become a member function of the new class is called a Member-Function-To-Be (MFTB). For all three conversion types, the brief of this application is that the CTB and all MFTBs are to be specified by the user.

Next, the most obvious conversions needed for Type I programs, will be demonstrated, and these will form a background for more detailed discussions in later sections. This will be done with reference to the small (invented) example program shown below, which uses a struct to represent a date.

```
1         struct date_tag {
2             int day, month, year;
3         }
4
5         int inLeapYear(struct& date_tag date) {
6             if (! (date.year % 4) == 0)
7                return 0;
8             else {
9                if (.....
10        }
11
12        main() {
13            struct date_tag today;
14            read(&today);
15            if (inLeapYear(today))
16               printf("It is a leap year!\n");
17        }
```

The transformed version of the program is shown below.

```
1         class date_tag {
2            public:
3             int day, month, year;
```

```
4           int inLeapYear;
5       }
6
7        int date_tag::inLeapYear() {
8           if (! (year % 4) == 0)
9              return 0;
10          else {
11             if (.....
12      }
13
14      main() {
15          date_tag today;
16          read(&today);
17          if (today.inLeapYear())
18             printf("It is a leap year!\n");
19      }
```

The first essential change is the creation of the new class, as is shown by the change of lines 1-3 in the original program into lines 1-5 in the new program.

```
struct struct-tag { ..... }
```

is to be replaced by

```
class class-name {  ......
                      MFTB prototypes }
```

Secondly, inside member functions references to member variables of the CTB argument object are replaced by just the name. In the example, line 6 is transformed into line 8, and in general:

```
CTB-argument. member
```

becomes

```
member
```

Finally, in the body of non- MFTBs, calls to MFTBs are to be changed in the following manor, as can be seen by comparing line 15 in the original program with line 17 in the converted program:

```
MFTB-name( .... CTB-argument )
```

is converted into

```
CTB-argument. MFTB-name( .... )
```

Outside MFTBs, there is no need to update accesses to member variables, as these should not be affected.

### *3.3. Issues relating to Type I*

#### 3.3.1 Non - & arguments

For the standard conversions above, it has been assumed that the CTB argument (i.e. the variable date) is passed by reference. For member function calls, the receiver object is (implicitly) passed by reference, so when the struct variable becomes an object, it will be passed in the same way. However, if the CTB variable is originally passed by pointer, or by value, it will be passed in a different way in the new program. The implications of this will be examined below.

An example of passing an argument by pointer is given in the following code extract from a dynamically allocated stack, given in Kruse *et al* [11]. The complete code for this can be seen in section 6.12.1. Here the CTB is Stack_type, and the function adds node_ptr to the top of the stack.

```
typedef struct node_tag {
    Item_type info;
    struct node_tag *next;
}  Node_type;

typedef struct stack_tag {
    Node_type * top;
}  Stack_type;

void PushNode( Node_type *node_ptr,
                              Stack_type *stack_ptr) {
    node_ptr -> next = stack_ptr -> top;
    stack_ptr -> top = node_ptr;
}
```

For pointer arguments, the conversion to a reference argument makes little difference inside the function. The conversion is changed so that:

```
CTB-argument-> member
```

becomes

```
member
```

Also, the CTB variable used to call the function may be a pointer, in which case:

```
MFTB-name( .... CTB-argument )
```

becomes

```
CTB-argument-> MFTB-name( .... )
```

These are shown in the converted function below.

```
void Stack_type::PushNode( Node_type *node_ptr) {
```

```
    node_ptr -> next = top;
    top = node_ptr;
}
```

An example of a value argument is the following access function for a vector composed of integers, converted from a student exercise written in Pascal, and shown in full in section 6.12.3.

```
int heap[NUM_ELEMS];

typedef struct vec_tag {
    int first;
    int length;
               }  vector;

int index(vector v, int i) {
    if ( (1 <= i) && (i <= size(v)))
        return heap[v.first + i - 1];
    else
        return 0;
}
```

Here, problems occur if any modifications are made to the CTB argument. In the original program, these modifications would not affect the variable used when calling the function - but in the object oriented version, they would. Therefore, the following conditions need to be satisfied before the function can become a member:

1. the CTB argument is not assigned to
2. the CTB argument is only passed to const member functions, or to functions that take it as a value parameter
3. no member variables of the CTB argument are assigned to
4. data members of the CTB argument are only passed to functions that take them as value parameters

The member function produced could then be declared as a const member function, which is semantically equivalent to having the class object as a value argument. The code is shown below, and in fact in this case, because an external heap is used, the const modifier is only useful as extra documentation.

```
int Vector::index(int i) const {
    if ( (1 <= i) && (i <= size) )
        return heap[first + i - 1];
    else
        return 0;
}
```

### 3.3.2. CTB return type

If a function returns a variable of the CTB type, but has no CTB arguments, then when it becomes a member this returned variable should become the receiver object. In fact, on examining more examples, it became clear that

fundamentally any function returning a CTB type should become a constructor function.

This CTB can be returned 'by value' in the original program, as shown in the example below, again from the vector exercise.

```
vector newvec(int s) {
    vector newV;
    if (s >= 1) {
        newV.first = heapPtr;
        newV.length = s;
        heapPtr += s;;
    }
    return newV;
}
```

Here, modifications in addition to changing references to members are required. This is because the object returned is usually constructed using a local variable, and setting the fields of this before returning it. The required modifications are therefore to remove firstly the return statement, and secondly the declaration of the variable returned, as shown in the constructor version of ‘newvec’ below.

```
Vector::Vector(int s) {
    if (s >= 1) {
        first = heapPtr;
        length = s;
        heapPtr += s;;
    }
}
```

There is also a non - sequential element to this kind of conversion, as first of all the identity of the variable returned must be found from the return statement. Then the declaration of this variable must be removed, and its declaration must occur before it is used in the return statement. References to its members must also be changed, and so the transformations of the early lines in the function depend on information gained from (probably) the last few lines.

The case where a pointer to a CTB variable is returned is particularly common in C, and an example of this is given below. This is taken from a queue type queueCDT given in Roberts [10] (shown in full in section 6.12.2).

```
struct queueCDT {
    void *array[MaxQueueSize];
    int len;
};

queueCDT * NewQueue(void) {
    queueCDT *queue;
    queue = New(queueCDT *);
    queue->len = 0;
    return (queue);
}
```

With this type of function, there are even more difficulties. Again the declaration and return of the "queue" variable have to be removed, but as is usual for a C initialisation function returning a pointer, it also allocates the memory for the object. This is of course not necessary, desired or possible with a constructor function, and so ways of removing the memory allocation statements were investigated.

In C, memory allocation is performed using the malloc() and related functions, but these functions are not always favoured by programmers (they have been superseded in C++). Therefore, as in the above function, it is fairly usual to find higher level functions used instead, which themselves call malloc, but handle errors better or have more understandable calling signatures. In the above example, this is the ‘New’ function.

This means that removing statements of the form

```
(CTB-variable) = malloc(...);
```

will not work in general, so the best solution found currently is to remove any statement of the form

```
(CTB-variable) = .........;
```

This could remove useful statements, but should work in the majority of cases. The modified version of the queue example is shown below.

```
QueueCDT::QueueCDT(void) {
    len = 0;
}
```

However, it is also common to see statements of the form

```
if (((CTB-variable) = malloc(....)) == 0)
    error("Out of memory");
else {
    .........
```

An example, taken from C-tree [14] (file typetab.c), is given below.

```
TypeItem *tptr;
bckt = symhash(sym) % MAX_HASH_BCKTS;
if ((tptr = (TypeItem *) malloc( TYPE_ITEM_SZE )) ==
NULL)
     return(0);
tptr->type_nme = sym;
tptr->node     = node;
```

For these examples, removing the assignment statement as above will result in an empty if-test-expression, which is syntactically incorrect. The solution here is to replace all of the if-statement by the else part only, for all if- statements with an assignment to a CTB-variable in the test. This seems to be even more drastic and prone to removing vital sections of code, but no examples have yet

been found when it would do so and examples have been found where not removing it would lead to incorrect code.

Functions with more than one return statement are also common in C, as the execution of a function terminates as soon as the first return statement is reached. However, no examples were found where this happened, and it is probably not nearly so convenient for functions that return a structured type. If such a case was found, the approach would be to remove all of the return statements, and if the resulting code was not quite correct, the user would have to rectify it.

### 3.3.3. Protect Specification

As the CTBs original member variables are declared in a C struct, they are visible in the same scope as the struct, i.e. they have 'public' protect specification. However, good software engineering practice dictates that the availability of variables should be as restricted as possible, and ideally all variables belonging to a class should only be accessed from within member functions.

Whether this is the case depends on how the original program was written, and which functions are to become members. The simplest course of action, and the only one guaranteed not to cause problems, would be to have all the member variables of the new class declared as public.

However, to maximise the benefits of the conversion to object orientation, the best approach is to check where each variable is accessed from, and if accesses are restricted to member functions, then to declare that variable as a private member of the class.

### 3.3.4. Typedefs

In C, when a struct is declared as in the example below (again from the stack given in Kruse *et al* [11]), the "stack_tag" represents a tag rather than, as would be expected, a new type. This means that variables must be declared as "struct stack_tag *var-name*", and this adds further complications to the conversions.

```
struct stack_tag {
    Node_type  * top;
};
```

The usual way used by C programmers to avoid this inelegance is to create a typedef for the struct, as shown below.

```
typedef struct stack_tag stack_type;
```

There are actually several different ways of creating this typedef, for example it can be combined with the definition of the struct:

```
typedef struct stack_tag {
    Node_type  * top;
}    stack_type;
```

Now, the tag is actually not needed:

```
typedef struct {
    Node_type  * top;
}    stack_type;
```

And finally, pointers are used extensively in C and so the typedef is sometimes for a pointer to the struct, as shown below.

```
typedef  struct  stack_tag * stack_pointer;
```

These mean that the program must recognise several names as referring to the CTB, so that function arguments with type "struct stack_tag", "stack_type", "stack_pointer", or "stack_type *" would all be treated as CTB variables. Also the program must cope with slightly different syntactic forms of struct definitions.

The typedefs themselves should remain in the transformed program, in case they are used by functions that will not become members to declare the CTB type. However, the reference “struct *CTB_tag*” should be changed to “class *NewClassName*”.

Another issue is the name of the new class, which ideally should be "stack_type" in the example above. However, an equivalent typedef will not be present in all programs to be transformed, so this is not seen as the best solution. As the user will anyway have to identify which class in the source program is the CTB, it is thought best that whatever name for the struct is supplied by the user is used as the name of the new class.

### 3.3.5. Multiple CTB arguments

If more than one CTB variable appears in the function header of an MFTB, it has to be decided for which, if any, the function is to be a member.

An example of this is shown below. This is a general (Pascal type) list class.

```
typedef struct node_tag * List;

typedef struct node_tag {
    int item;
    struct node_tag *next;
} ListNode;

List cons(List l, int i) {
    List temp =(List) malloc(sizeof(ListNode));
    temp->item = i;
```

```
    temp->next = l;
    return temp;
}
```

It is not immediately obvious which list should be the receiver object. If it was a method of the argument rather than returned list, it would be returning a new list that was itself, with an extra item on the front. If it was a method of the returned list, it would be a constructor combining an item and an old list, and this interpretation seems to make slightly more sense.

It is possible that for functions with one CTB as a return type, and one as an argument, it is always logical to make them into constructor functions. However, it is felt that insufficient examples have been analysed to say this for certain.

It is also possible to have more than one CTB argument, as the following function prototype for a hypothetical complex number class shows.

```
void add(complex arg1, complex arg2, complex &result);
```

Here it is very hard to say which object should be the receiver. It would be reasonable to declare the function as a friend to the complex class, although this may be seen as defeating the point of having it as a member function. Alternatively, the first variable declared in the argument list could be taken to be the receiver object, or the information could be specifically requested from the user.

With further analysis of a considerable number of examples, it may be possible to devise a systematic method for determining the best course of action. However, there do not seem to be any language features which can be used to indicate which object is the most logical choice to become the receiver. Ctrans will refuse to convert any functions with more than one CTB declared in their header, but for a future update, it may be better to consult the user.

### 3.3.6. Calling MFTBs

It is also necessary to modify the calls to MFTBs, both from MFTBs and from non- MFTBs. Generally, this is straightforward, with the argument that is a CTB variable being removed from the argument list, and the call to the function being made on this object via either a "." of a "->" operator. In the example from section 3.2,

```
if (inLeapYear(today))
```

becomes

```
if (today.inLeapYear())
```

However, this is not the case inside MFTBs if the CTB-argument is the same variable that is passed to the calling MFTB. Then, the only action taken is to

remove the CTB-argument from the argument list, as is demonstrated below, where ‘getDayNumber’ is also an MFTB.

```
int getDayNumber(struct date_tag & date) {
    if (date.month > 2) {
       if (inLeapYear(date))
          ........
```

This is changed to:

```
int date_tag::getDayNumber() {
    if (month > 2) {
       if (inLeapYear())
          ........
```

For MFTBs that return a CTB variable, a different action is required, as these will be converted into a constructors. If the CTB variable returned is not a pointer, then the name of the MFTB is replaced by the name of the new class, which is the name of the constructor. However, if the variable is a pointer, it is likely that in the original C program this call allocated the memory for the object.

In the converted version, any memory allocation inside the MFTB will have been removed, and so it will be necessary to allocate memory for the object as well as calling the constructor. This can be done simply by replacing the call to the MFTB by "new *NewClassName*", as shown below. This will work even if the result of the call is not assigned immediately to a variable, but, for example, passed directly to a function.

```
struct date_tag *new_date;
new_date = read();
```

This will become:

```
date_tag *new_date;
new_date = new date_tag;
```

In both cases for the constructor MFTBs, all of the arguments remain the same.

## *3.4. Issues relating to Type II*

### 3.4.1. Introduction

For Type II programs, it can be argued that as they deal with a group of variables as a unit, but do not define a struct to store them, they must be fairly badly written.

However, part of the purpose of the tool is to undo the effects of hasty programming, and give a better organisation to programs. Also, at a higher level,

conversions of this type could be useful for creating large interface or control objects that would not exist in the imperative version.

Many of the issues examined for Type I can be extended to this case; for example, if an MFTB has any of the variables making up an object as value parameters, then it must not modify those variables. Some aspects that are unique to Type II are discussed in section 3.4.2 below.

### 3.4.2. Conversions

The only example found of a Type II program is from Hanly *et al* [9], and it is a simple program that performs calculations on fractions. An idea of the working of these functions can be gained from the (considerably edited) code given below.

```
void scan_fraction(int *nump, int *denomp) {
    int numerator;
    char delimiter;
    int denominator;
    printf("Enter a fraction\n");
    scanf("%d%c%d", &numerator, &delimiter,
                                &demoninator);
    if (delimiter != '/')
        Error("Wrong format for fraction");
    *nump = numerator;
    *denomp = denominator;
}

int main(void) {
    int n1,d1;
    int n2,d2;
    int n_ans, d_ans;
    scan_fraction(&n1, &d1);
    scan_fraction(&n2, &d2);
    add_fractions(n1,d1,n2,d2,&n_ans, &d_ans);
    print_fraction(n_ans,d_ans);
}
```

As there is no obvious aggregate data structure in this case, the new class must be specified in terms of the member variables that it is composed of. The user can define these by entering the type and name of each individually. Additionally, users may find it convenient to be able to define the member variables as being the same as the formal parameters of a particular function; for example, the function "print_fraction" in the example above actually has "int numerator, int denominator" as its arguments. The name of the new class must also be entered by the user.

Similarly, when an integer is used within an MFTB, there is no way of automatically determining whether or not this integer corresponds to for example the "numerator" member of the fraction class. The user will therefore have to specify which variables correspond to which members of the new class, both in the argument line of MFTBs, and whenever variables declared in the program should be replaced by an object declaration.

The appropriately modified code is shown below. Note that the friend solution has been used to the multiple CTB argument problem.

```
class Fraction {
    private:
        int numer;
        int denomin;
    public:
        void scan_fraction();
        void print_fraction();
        friend void add_fractions(Fraction f1,
                        Fraction f2, Fraction &f3);
}

void Fraction::scan_fraction() {
    int numerator;

    <unchanged>

        Error("Wrong format for fraction");
    numer = numerator;
    denomin = denominator;
}

int main(void) {
    Fraction f1;
    Fraction f2;
    Fraction f3;
    f1.scan_fraction();
    f2.scan_fraction();
    add_fractions(f1,f2,&f3);
    f3.print_fraction();
}
```

### 3.4.3. Problems

There are several problems with this type of conversion that have not been adequately solved yet. A possible difficulty is that names have to be created for the objects declared in the converted program. Names could be automatically generated for these, for example *NewClassName*1, *NewClassName*2, and so on, but this may detract from the readability of the program. Alternatively, the user could choose names for any new objects, but ideally this would be avoided, as the user already has to specify several new names. Also, with both these names and the names for the member variables of the new class, there is a possibility of clashes with existing variables.

The other problems concern the calling of MFTBs. The first of these is that, although the formal parameters of the MFTB may clearly constitute an object of the new class, the actual parameters used may not. This is because the actual parameters need not be variables, as in the example below.

```
print_fraction(2,get_int());
```

This rules out the ideal scenario, which is to replace the declaration of two integers with a Fraction object, and use this object in place of the integers throughout the rest of the program. For this kind of call, a solution such as indiscriminate generation of objects is needed, which reduces considerably the advantages of producing an OO program. Such a solution requires firstly a constructor function for the new class to be automatically created, which will initialise all the members to values given. This constructor could then be called to create a new object just before the call to the MFTB, and given the same arguments as the MFTB was in the original program, as shown below.

```
Fraction fraction8(2,get_int());
fraction8.print_fraction();
```

Another serious problem occurs when the actual parameters of the MFTB correspond to different members in the new class at different times. The example from the fractions program is the calling of multiply_fractions when the desired result is division. This is achieved by passing the numerator as the denominator and vice versa for the second argument fraction:

```
          :
          :
else if (operator == Mul)
    multiply_fractions(n1, d1, n2, d2, &n_ans, &d_ans);
else if (operator == Div)
    multiply_fractions(n1, d1, d2, n2, &n_ans, &d_ans);
```

Here there should in fact be two separate functions, and there is no way that this could be recognised or dealt with by an automated tool.

In conclusion, there are many difficulties with Type II conversions, which do not exist in Type I. Also programs suitable for Type I conversions appeared to be much more common than those suitable for Type II conversions, and so it was decided to concentrate on the Type I transformations.

### *3.5. Issues relating to Type III*

This type of conversion differs only slightly from the Type II transformations. For a Type III conversion, the CTB is again a set of variables, but these are declared globally and therefore do not appear as arguments of the MFTBs.

The example is a student exercise, for producing an encrypted version of a text file using a lookup table containing the alphabet in a different order. It uses a global array of char, created with the use of a special keyword, and a few extracts are given below.

```
#include <stdio.h>

const char alpha[] = "abcde...";
char ciph[27];
```

```
void MakeTable(char * keyword) {
    for (i = 0; i < strlen(keyword); ++i)
        strcat(ciph, keyword[i]);
    ......
}

char Encode(char ch) {
    int posn = GetIndex(alpha, ch);
    return ciph[posn];
}
```

This is a much simpler case than Type II, with very few modifications being needed to the code, as the references to members of the new class within member functions will not be changed. The only significant changes are to create the class and declare (globally) an object of the new class. Also, calls to the MFTBs have to be converted to use the global object. The transformed code is given below.

```
#include <stdio.h>

class Table {
    private:
        const char alpha[] = "abcde...";
        char ciph[27];
    public:
        char encode(char ch);
}

Table table;

Table::Table(char * keyword) {
    for (i = 0; i < strlen(keyword); ++i)
        strcat(ciph, keyword[i]);
......
}

char Table::Encode(char ch) {
    int posn = GetIndex(alpha, ch);
    return ciph[posn];
}
```

Here, the user has to specify, in addition to the constituents of the new class and the MFTBs, the name of the object(s) of the class, but not any associations between formal parameters and the members of the new class. The global nature of the data members of the new class means that none of the other Type II problems occur.

It is necessary for the user to indicate if any MFTBs should become constructor functions, as for example the MakeTable function above logically should be, because there is no way of recognising this automatically.

Again, the style used in a program suitable for Type III conversions is probably very bad, and it is not expected that many Type III transformations would be encountered.

### *3.6. Requirements for semantic processing*

In the transformations outlined above, certain knowledge about the code being transformed has been assumed, for example the type of any identifier used. Such information will have to be found explicitly by the tool, and the exact nature of this information is described below.

Firstly, some details about the top level declaration each statement occurs within are needed. This is because statements that appear in an MFTB body are transformed differently from those in a non-MFTB body, or those in a struct definition, and so on. For example,

```
if (inLeapYear(today))
```

becomes

```
if (inLeapYear())
```

within a member function, but

```
if (today.inLeapYear())
```

in a non-member function.

Secondly, an important question is whether the name of an identifier is enough to identify it uniquely, and this relates to issues of names and scope. In general for C the name is not enough, as identifiers can be re-declared in an inner scope, and so names used in the input program need to be resolved by some sort of semantic analysis.

The first case where this may be a problem is in the function arguments, where it has to be determined if the name used to specify the type of an argument really represents the CTB. In this case, there is actually no problem, because functions in C can only be defined in the outer scope. This means that any typedef- name used for function arguments must also be declared in the outer scope, and if this new type had the same identifier as the CTB there would be a name conflict, as in the following example.

```
typedef struct date_tag {
    int day, month, year;
} Date;

typedef long Date;   /*  Not Allowed!! */

int inLeapYear(Date& date) {
    if (! (date.year % 4) == 0)
        .....
}
```

However, some type information is needed for function arguments, as the conversions are different for reference, pointer or value CTB arguments. For example, for reference or value arguments, statements of the form:

```
CTB-argument. member
```

become

```
member
```

whereas for pointer arguments, the same conversion must be applied to statements of the form:

```
CTB-argument-> member
```

A related point is that only structs defined in the outer scope can be transformed, as otherwise no functions can take a variable of the struct type as an argument.

It is also possible for several different functions to have the same name in C++, as long as they have a different number of arguments and/or different argument types (function overloading). Therefore the name of a function is not enough to identify it positively as an MFTB, and other checks need to be made, for example on the arguments used. These extra checks are needed both when the function is defined and when it is called.

Type information is required within functions, where it is necessary to know if an identifier's type is really the CTB. This is so that, in function calls, the variable can be recognised as a CTB. There are two cases where the type of an identifier could be confused, as described below.

Firstly, it may be possible for the identifier of the CTB argument of an MFTB to be re-declared within a local scope as a different type. Some compilers will complain that a parameter is being "shadowed" if this is found, but the front end actually used for this project accepted it. Therefore, it was felt best to always check the type of any variable, even if it has the same name as the CTB argument, before any transformations are performed.

```
int inLeapYear(Date& date) {      /* CTB argument */
    if (! (date.year % 4) == 0)
       int date = getNum();
       if (isOdd(date)) {      /* NOT a CTB var  */
       .....
}
```

Additionally, it is possible for the type of a variable declared within the function to appear to be the CTB, but actually be something else. This could happen if a typedef for the CTB name was declared, so that it represented some other type, within an inner scope. Then a variable could be declared in that scope, and the name of its type would be the same as the CTB type-name.

```
typedef struct date_tag {
    int day, month, year;
} Date;
```

```
int inLeapYear(Date& date) {
    typedef long Date;
    if (! (date.year % 4) == 0)
       Date not_ctb_var;   /* This is a ‘long’ variable*/
       if (isOdd(not_ctb_var)) {
       .....
}
```

### *3.7. Summary of program requirements*

The basic requirements for the tool are for it to take an input program in C or C++ and convert data structures and functions in that program into objects and their associated methods.

This conversion is to be user directed, and so the tool must display the input program text and enable the user to specify which structures and functions are to be converted.

There are three basic kinds of conversion that the program should perform, corresponding to the three types discussed above:

1. Turning a C struct into a C++ class, and turning specified functions from the C program into members of the new class.
2. Creating a new class from a set of variables that are generally used together in the original program. These variables may be specified by (a subset of) the arguments of a particular function, or by a set of variables declared in the program, or entered explicitly.
3. Creating a new class to contain a set of global variables that are manipulated by functions in the program, and making these functions class methods.

These conversions can be performed in stages, for example by specifying a C struct to be made into a class, or by specifying a variable or function to be added to an existing class. However, batch conversions should also be allowed where a struct to be converted to a class is given along with a list functions to become class methods.

Finally, the tool will require the input program to be correct.

# 4. DESIGN

## *4.1. Syntax tree*

### 4.1.1. Background

When compilers convert a program into machine code, they require a representation of the input program so that its structure is unambiguous, the order of applying operators is explicit, and so on. Compilers create a syntax tree to provide this representation.

This tree is created in a sequence of phases, the first of which is the lexical analysis phase. This splits a program into a series of tokens, each of which is a distinct syntactic unit such as a number, a keyword of the language, or an identifier.

The next phase is syntax analysis or parsing, where tokens received from the lexical analyser are grouped together to form statements in the language, and appropriate representations of these are added to the syntax tree.

Next, usually as a separate pass over the syntax tree, semantic analysis is performed. This stage is concerned chiefly with determining and ensuring consistency of types, for example checking that expressions have the same type as the variable they are assigned to.

Finally, there are several phases a compiler uses to generate the output machine code, but these are of little relevance to the project and will not be described here.

Parsing usually performed by referring to a Context- Free-Grammar which specifies the syntax of the language. This grammar consists of a set of rules for obtaining the non- terminals of a language (function definitions, while statements, binary operand expressions,....) from the terminal symbols (ones that can not be split up any further - identifiers, or keywords such as ‘if’, ‘char’).

An example of a syntax tree is shown in figure 4.1 b), with the code producing that tree shown in figure 4.1 a). This example uses the same node types and syntax as the Cppp parser, discussed in section 4.1.2. below.

Finally, compilers will also use tables in conjunction with the syntax tree, often a symbol table and a separate type table which will record details of all the user- defined types, and all types of user defined variables and functions in the program. The symbol table will have an entry for every identifier used, and will usually store information such as its name (a string), a pointer to its type, (if it is a function) its arguments, and where it is used in the program.

a)

```
int isGreater(int a, int b) {
   if (a > b)
      return 1;
   else
      return o;
}
```

b)

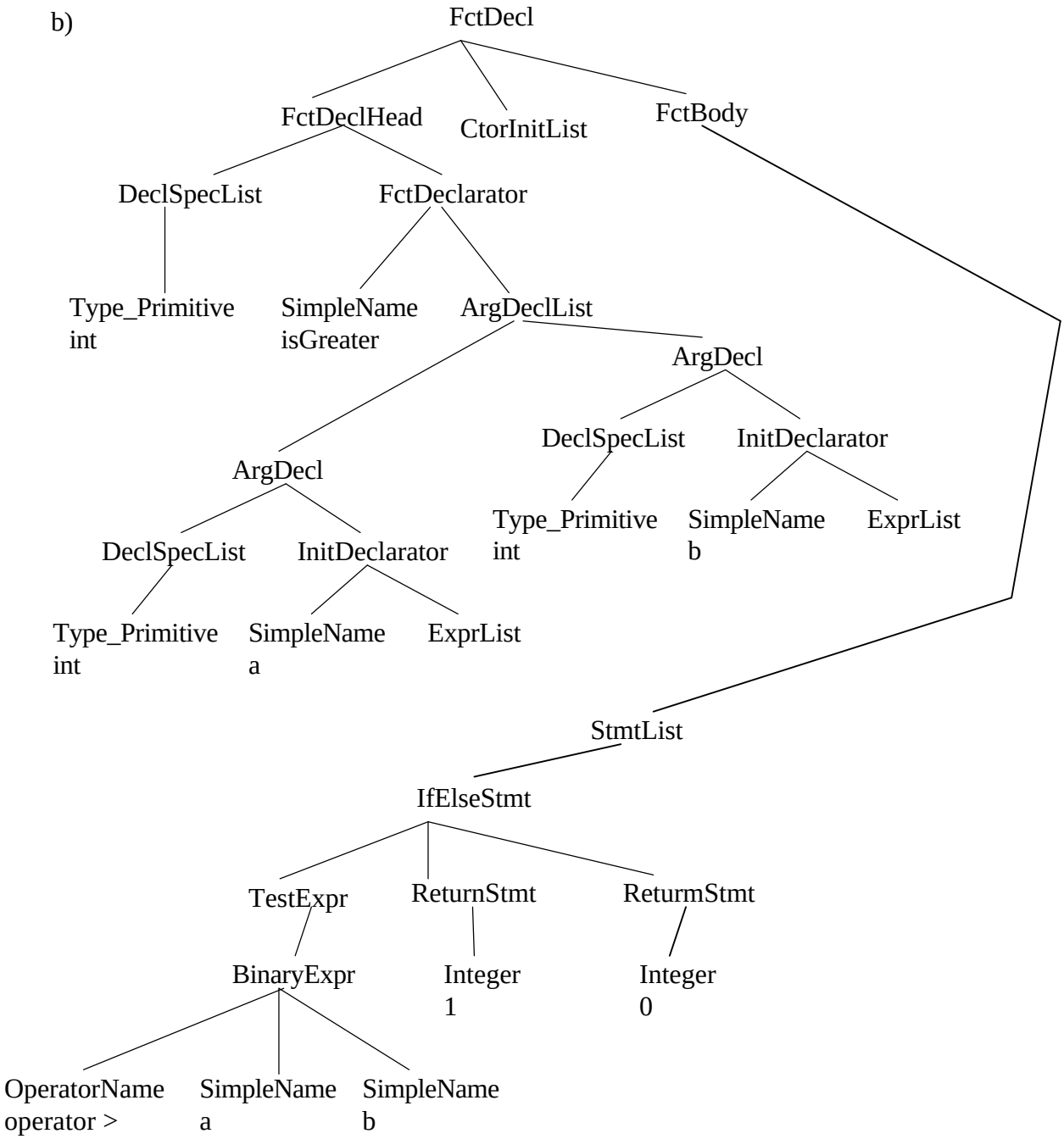



**Figure 4.1. Example program and syntax tree**

More can be read about compilers and trees in the definitive book by Aho, Sethi and Ullman [12], or in Bennett [13] which is a good brief introduction.

### 4.1.2. Tools

Many tools exist to support parser generation, partly because compilers are written so often (one of the most common tools is called Yacc, which stands for Yet Another Compiler- Compiler). However, there are many other reasons for wanting to translate text written in a certain format, or according to certain grammar rules, into its syntactic components. Examples are a calculator processing text input, or a program to read in a table of figures.

The "standard" parsing tools are Lex and Yacc, which are available on almost all Unix environments. Lex takes as its input a specification of the keywords, format for numbers, rules for identifiers and so on of a language. It produces as its output a C function that will split a program into a stream of tokens, according the rules supplied.

Yacc is the parser complement to Lex, and it takes as input firstly the grammar defining the language, and additionally a set of actions (written in C) to perform when each non- terminal is recognised in the input. For example, if a non-terminal

*if-stmt*

is found from the components

if (*expr*) *stmt*;

then the action could be to produce a syntax tree node with the *expr* as its left child and the *stmt* as its right child.

Lex and Yacc are used by most compilers, and many parsing tools using a Yacc grammar are available, often free of cost via the Internet. The tools that were considered for this project are the C-tree c-parser, the Roskind C++ grammar, the GNU g++ compiler and the Cppp C++ parser/semantic analyser. A brief summary of each of these is given below.

The first tool found was C-tree [14], which parses and creates a tree for C programs, but not for C++ programs. It includes fairly straightforward data structures and building procedures for the tree, and has facilities for printing out a representation of the tree, and also a rudimentary version of the tree as code.

A C++ grammar written by Roskind [15] was also obtained, which is a well documented and clearly written grammar, but it describes a slightly out of date syntax (for example, no templates are allowed). It is also just the Lex and Yacc input files, without the associated actions required for building a syntax tree.

More complete packages were also looked at, firstly the GNU g++ compiler from the Free Software Foundation [16]. This is written in C, but being a

full compiler package, it parses the current version of the C++ language totally reliably. It also performs semantic analysis, type checking, name resolution, and contains modules to generate the machine code version of the program. These separate features, however, appear to be well integrated into the general structure.

The g++ program is in fact combined with the gcc C and objective-C parser, and is a very large piece of software, with literally hundreds of separate files and approximately 500,000 lines of code. The code does contain comments, but overall the package is quite badly documented.

Finally, a package called Cppp (for C- Plus- Plus- Parser) was found [17], which as well as parsing C++, is written in object oriented C++. This is, like the C-tree program, designed to be a front end tool and to be incorporated into other programs, and has a feature to output a text (but not code) representation of the tree. Additionally, unlike C-tree, it performs semantic analysis of the input program.

Cppp is a fairly large program too, but has some documentation of its structure. According to the authors, “the C++ front end is still in a prototype form” [17], but it parses an up- to-date version of the language, and even has an additional lookahead functions to resolve language ambiguities.

### 4.1.3. Structure of ctrans

It was obvious that the tool would need a lexical analysis stage, to split the program into tokens, but the necessity of a syntax tree was not so apparent. Therefore the first key design question was whether the output OO program should be produced from an abstract syntax tree, or if a sequential pass over the input program would do.

The obvious drawback of using a syntax tree is that a foreign piece of software will have to be incorporated into the tool, or part of a parsing program will have to be written. In addition to the problems likely to result from this, a key disadvantage is that some information contained in the input program will be lost, and this information is described below.

- The layout of the code would be lost, which could be very irritating to a programmer who had spent some time indenting and spacing the code to his/ her liking, and then found it looked very different after being transformed.
- Any pre-processor macros used in the source program would have to be expanded before parsing, and would not appear in the modified code. This could make further changes to the program considerably more time consuming, as macros are often used to produce many sections of fairly similar code.
- The comments contained in the code would be lost, and for many programs this would have a detrimental effect on their maintainability that could not be compensated for by any structure change.

This is not information that will change the meaning of the program, but it is important for the readability and maintainability of the program - and improving these aspects is the goal of the tool.

Parsing the whole program also seemed slightly wasteful, as only some sections of it were needed for the transformations - struct definitions, the headers of functions and function calls, and accesses to struct members. Therefore, the initial idea was to only extract the names and types of variables, building a symbol table, on a preliminary scan of the input program. The conversion would then be performed on another pass over the input code, copying directly to an output file all sections except the statement types that may need converting.

The key problem with this is that to identify the required sections, for example calls to functions or a function definition, the tool would be essentially parsing the input anyway. Also, more complete information would have to be stored about functions, as function prototypes for the MFTBs must be incorporated into the definition of the new class. As this definition must replace the struct definition, the output for the MFTBs would have to be produced from a syntax-tree like representation of the function headers.

These problems could all be solved, albeit in an unsatisfactory manner, but the semantic information, required to identify uniquely functions and variable references, was still impossible to obtain without a syntax tree. The penultimate design, still not wholly dependent on a syntax tree, was to perform all the conversions on a syntax tree, but create "triggers" for each conversion required. These triggers would consisting of the line number in the source code and first few tokens of the original code fragment. The input file would then be copied to the output until a trigger was encountered, when the modified version from the syntax tree would be written to the output instead.

However, it was realised at this stage that the tool would have to be able to output a large proportion of the tree as code (class definitions, function headers, variable references), and it would be far easier to output the whole target program from the syntax tree.

This also has the advantage of consistency - the program can be 'pretty- printed' - and there would be no complications such as having to keep track of line numbers in the input program. The lack of comments is not so important for a demonstration prototype, as this tool would really be, and even without a syntax tree it is hard to see how the pre-processor commands could have been kept.

### *4.2. Choice of Cppp as front end parser*

The options for the tool to use to create the parse tree were as follows:

1. Create a grammar for a subset of C++, use Lex and Yacc to write the parsing program, and also write a semantic analyser.

2. Disentangle the parsing and semantic analysis sections of g++ from the rest of the program.
3. Build a tree generator (probably similar to c-tree) on to the Roskind C++ grammar, and write a semantic analyser.
4. Use Cppp as it is.

Given these choices, the final decision to use Cppp seems obvious. The main problems with g++ were the difficulty of isolating and understanding the components that would be needed, given the scale of the program, and the non-object oriented nature of the syntax tree. The Roskind grammar was perhaps a more attractive option, where the coding could be started from scratch. Even then, there was a risk of encountering problems with parsing C++, and a big risk of difficulties with finding the required semantic information.

However, there were also significant problems with Cppp. The first of these was that, despite a fair amount of effort having been already spent on the problem, the program would not compile on the local Unix environment. It was felt that this could be fixed, as Cppp had been tested in a similar environment, and that the time invested fixing it would have a considerable payoff.

Also, Cppp is a large piece of software, with utility classes defined for strings, lists, hash tables and so on, and extensive use of the pre-processor, even having a series of macros for creating new classes. The Cppp archive library is 2 ½ megabytes in size, and this scale meant that becoming familiar with the program would be a not inconsiderable task.

Despite these, the benefits of Cppp meant that it was still considered the best option. These included easy access to the tree produced (both as a whole and via access functions provided for the individual nodes), having semantic analysis performed, the whole package being written in C++, and the intuitive relation of the Cppp syntax tree to the corresponding code.

## *4.3. Design of the conversion engine*

### 4.3.1. Approach

The basic structure was now established, and is shown in figure 4.2. The command interpreter and the code output function are not considered key design points, and are discussed in sections 6.9 and 6.3.3 respectively. The parsing and semantic analysis are performed by Cppp, so the main task remaining was to develop the conversion engine.

This would work on a syntax tree, so initially examples of real syntax trees were examined to see the nature of the conversions required on that level. This was done using the c-tree parser, as it had a comparatively simple (C only) grammar, and the method by which the tree was built was also easily understood.

At first, the c-tree grammar was used to analyse by hand how the parser would split up a program, and the actions examined to reproduce the tree that would be created. This proved to be extremely tedious, and so it was felt worthwhile to try to get c-tree to compile on the local Unix environment, which it

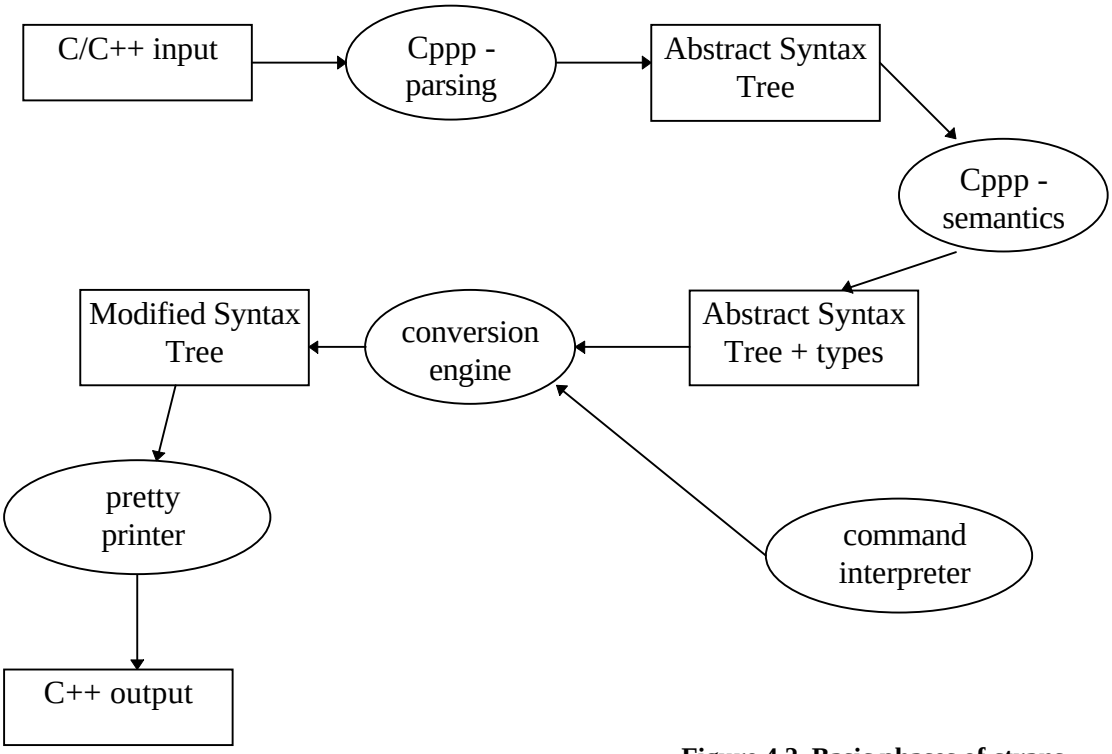



**Figure 4.2. Basic phases of ctrans**

had not done initially. This required relatively few changes to the makefile, and made it easy to produce a syntax tree for one of the original example programs (taken from a C textbook).

### 4.3.2. Alternatives: intelligent tree or pattern matching

Two alternatives were considered for the conversion engine: an ‘intelligent’, object-oriented syntax tree, and an algorithm based on finding and replacing patterns of nodes in the syntax tree.

The intelligent syntax tree will be considered first. As the project is based on the belief that OO code is better than imperative code, a requirement of the design for the heart of the program seemed to be that it was object-oriented. The main data items being manipulated are the nodes of the syntax tree, and so these are good candidates to become objects. With this approach, the base node class would have a method that instructed it to convert itself to the OO equivalent. It would respond to this by telling its children to convert themselves. The "convert" function would then be re-defined in certain node classes, for example the nodes representing access to a struct member (*struct-var-> struct-mem*), to perform the actual changes.

This approach is fairly natural, because the conversion of a node depends on its type. For example, the member access nodes would firstly have to check that they occur within a MFTB; if so, they would then check that their son representing the struct variable is the same one as the CTB-argument of the function. Having established this, the node would replace itself with just its son representing the member. This is as described in sections 3.2 and 3.3.1, and an idea of the required code is given below (using Cppp data and functions).

```
CpppAst_MemRefExpr::convert() {
    if (inside_mftb) {
         if (son(0)->astName() == “SimpleName”)
               if (son(0)->name() == ctb_arg_name) {
                     CpppAst temp = this;
                     this = son(1);
                     temp->freeNode();
               }
    }
}
```

However, it soon became apparent that with this design, nodes need to know a lot of 'global' data, especially about their parent nodes, and even sometimes about their siblings. There are also many conditions that need to be checked about the sons of a node, and depending on these conditions, there may be several different conversions for a node type. An example of the extent of these checks is the conversion required for every node representing a variable declaration, as given in section 3.3.2:

*If*

- within an MFTB
- return type is CTB *
- there are no CTB parameters in argument list
- the type specifier for declaration is CTB *, *OR* type specifier is CTB and a pointer declarator is declared
- only one variable is declared in this declaration
- this variable has the same name as the variable returned in the return statement in the function

*Then*

- remove self

Having each node check its ‘context’, and then decide how to convert itself, could lead to a disordered design. In the conversion procedures, the functionality would be hidden in the many checks on global variables, required to indicate the node’s current context. A non object- oriented approach would probably be worse, with a very large case statement in the main (recursive) convert procedure, needed to determine the type of the current node, in addition to the other checking code.

The main drawback of this kind of approach was felt to be the difficulty in adding extensions to the conversion engine. The analysis phase had shown that there are always several different ways of writing things in C, and there were still several scenarios that were not examined fully. For example, it was only

realised later in the development that all declaration statements with "struct *CTB-name*" as their type specifier would have to be modified to use the name of the new class. C is also a fairly old language, containing obscure structures like old- style function definitions, as shown in the example below.

```
int isGreater(a, b)
int a;
int b;
{
          if (......
```

On the other hand, C++ is a developing language, with relatively new features such as templates, and more changes anticipated with the C++ standards committee having recently made available a draft ANSI C++ standard [18].

Therefore, an important aspect of the design was felt to be that it should be easy to add new conversions to the ones already specified in section 3. This would not be the case for the intelligent tree as described above, where changes would have to be made at a fairly low level in the code.

An alternative design was therefore pursued, and an idea was developed from the examination of the example syntax trees. This was to encode the transformations required as a pattern, which has two parts; a tree of nodes to find in the input syntax tree, and a tree of nodes to replace them with. This approach was inspired by the pattern matching features found in functional languages, and also by the techniques described in Aho, Sethi and Ullman [12] for code generation.

A set of patterns would be data for the program, and the application itself would test patterns against the actual syntax tree. An example of a node pattern to find, and the nodes to replace it with, are shown in figure 4.3. Here a combined typedef and struct definition is converted into a class definition, with the same members, and the name of the new class is that of the original typedef. The nodes with a star and a number are an instruction to the application to keep a pointer to the node (and its sons) that appears at this place in the actual syntax tree.

The task that the program would perform would be to check if an if-find pattern matches a syntax tree (i.e. if the root node in the pattern is the same as the root node in the syntax tree, and all the sub-trees also match). The pattern nodes with star numbers match any node in the tree. If the whole pattern matches, the original syntax tree node is replaced by the pattern for the object oriented version.

This approach seemed quite simple, and would certainly allow the program to be extended to cover more conversion specifications only a new pattern would have to be written.

### *4.4. Choice of pattern matching*

The pattern matching approach seemed a very elegant solution, but there were problems with it. Firstly, it was not obvious that the system would be able to produce all the required modifications - there are many aspects of the conversions that are not easy to express in just the type of tree nodes. For example, in the definition of the new class, prototypes would have to be produced for the MFTBs, and access to information such as the name of the CTB for the current transformation would be required.

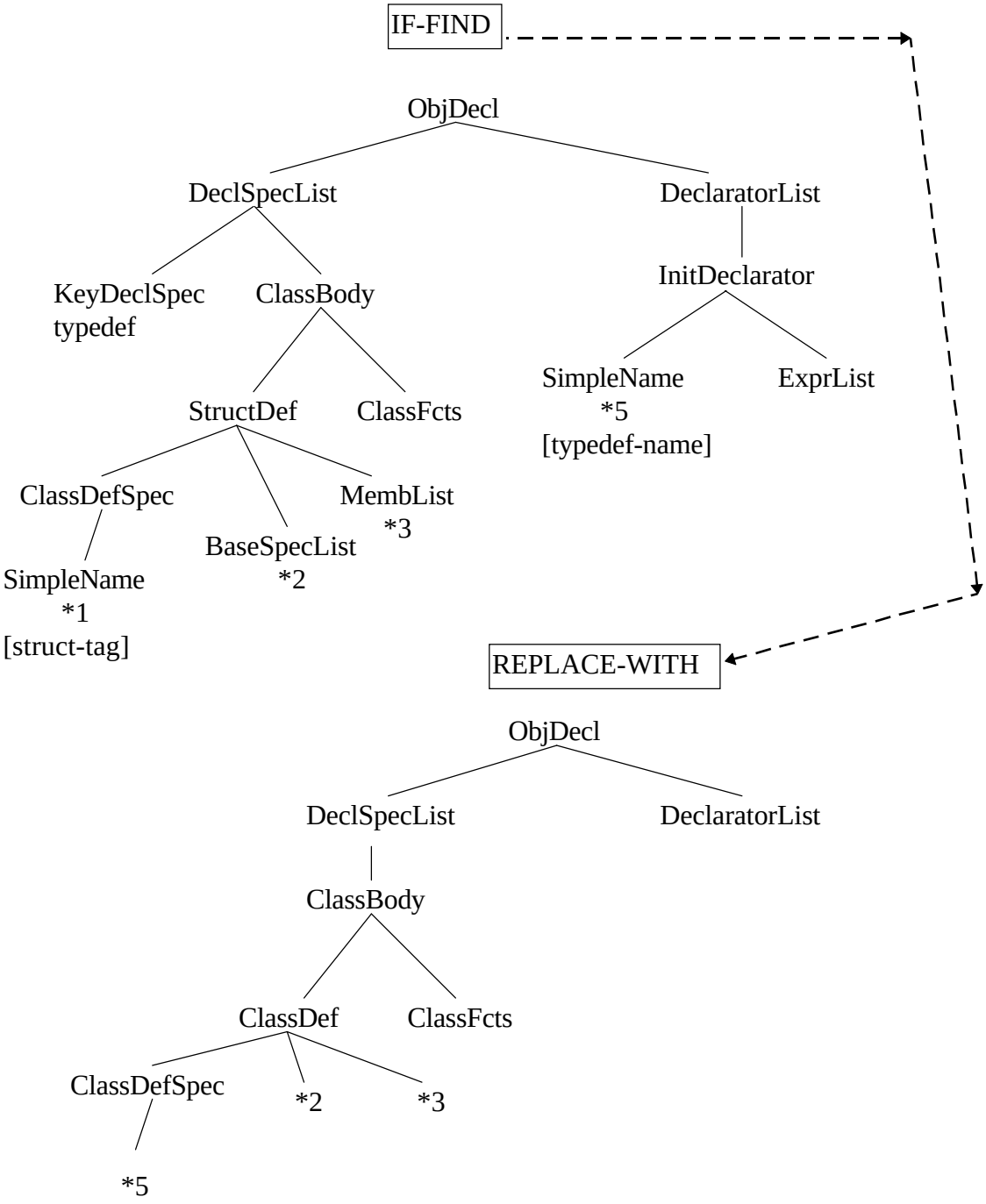



**Figure 4.3. Pattern for a typedef/struct definition**

Also the approach is not object oriented - it would probably be possible to represent the patterns as a set of objects, but the methods on them would have to be defined with the syntax tree as an argument. This is contrary to the object oriented philosophy that objects should modify themselves.

A further drawback is that each time a call to an MFTB is found, it must be converted as though it had not been seen before. It is not possible for the pattern to 'remember' a new calling template for the object oriented version of a function, even if this new argument specification had had to be inferred for a previous transformation.

However, for the final decision, the problems of extendibility and spaghetti design with the intelligent tree option were thought to be more important than these drawbacks of pattern matching. The choice of Cppp as the front end meant that the intelligent tree had another disadvantage, which was that the 'convert' member functions would require extensive modifications to the Cppp tree node classes. This would be contrary to the software engineering best practice of not making any changes to library code incorporated into new projects.

In summary, pattern matching was chosen because it was an elegant concept that made it apparently very easy to extend the program to cope with the complexities of C. Also, the concept was especially appealing because of the close correlation between the contextual information needed to determine how to transformation a node, and the information represented in a pattern.

# 5. PATTERNS

## *5.1. Basic Ideas*

### 5.1.1. Concepts

The essential concept of pattern matching was introduced as a means of describing program transformations in section 4.3.

The patterns come in two parts: an "if-find" pattern and a "replace-with" pattern. The if-find pattern is compared with the syntax tree to be converted, and if any section of that tree is the same as the if-find pattern, then that section is removed and the replace-with pattern put in its place.

Each if-find or replace-with pattern describes a section of syntax tree. It consists of a root node, and possibly several children, which in their turn may have children, and so on. The nodes have a label to indicate the type of statement or declaration they represent in the source program. For example a function declaration node is called FctDecl (in the Cppp tree), and must have a FctDeclHead, CtorInitList and FctBody as its child nodes.

The term pattern will be used to mean a specification for a complete transformation of a program, which will be defined more precisely latter. However, the term will also be used, qualified by ‘if-find’ or ‘replace-with’, to refer to a replace-with or if-find node together with all its sub-trees. This meaning will be used when the distinction between an individual node and a tree of nodes is felt to be important.

### 5.1.2. Stars

Normally, an if-find pattern matches a section of the syntax tree if the 'root' nodes of the pattern and the section have the same type, and the children of both match. However, a special 'star' marker can be used in a node of the pattern, which says that this node will match anything in the input tree. A number is also used with the star, and this becomes an identifier for that section of the input tree - the star number can then be used in a replace-with pattern, so that parts of the original tree can be kept in the modified tree.

This is still not sufficient to represent the conversions described in section 3, and the remaining deficiencies together with their solutions are described below.

### 5.1.3. Function headers

There is a problem with creating the function prototypes for MFTBs (in the new class definition), as a pattern for converting a struct cannot access the required information about the MFTBs. If the tree sections representing MFTB declarations were stored as star-numbers, the struct patterns could possibly use these. However, the struct must be defined before the MFTBs, and so these declaration nodes will not have been encountered and stored, at the point when the class definition has to be created.

The initial solution to this was to have a pre-pass over the syntax tree, which would record information about the MFTBs, and produce the new OO headers. These could be inserted in the correct place in the new tree by the pattern matching engine, using the trick of giving them a special star number (say 0). A better solution was in fact found later (see section 6.6.6).

### 5.1.4. Macros

The other obvious flaw is that a pattern is not specific to any one conversion, and if all programs could be converted in the same way, there would be no point in having any user input. The user input is the identity of the CTB and MFTBs, and this information must be made available to the patterns. This is achieved by a set of special "macros", which are added to certain pattern nodes as an extra condition on the source tree node for it to match (these were called macros because it was originally envisaged that each one would be a small predicate function). Example "special macros" or "special matches" are shown below, and a complete list is given in section 6.6.

- _MFTB_
  The syntax tree node must represent an identifier for an MFTB.

- _VarCTB_
  The node must be an identifier representing a variable whose type is the CTB.

- _DirCTB_
  The node must be a type name representing the CTB type (Dir => direct, as opposed to a pointer to the CTB).

### 5.1.5. Sub - patterns

The final basic problem appears when converting functions. The pattern must match the function header, but there is a difficulty with the function body, namely that most of the body should be kept as it is, so it should be marked with a star number; however, some bits of it need to be changed, especially accesses to data members of the CTB argument. It is not possible to write a pattern

to match the whole function body, and therefore only modify some of the nodes within this, and so a "sub-pattern" of if-find and replace-with is required. This, and example uses of the special match macros, are shown in figure 5.1 (just the if-find patterns are shown for clarity).

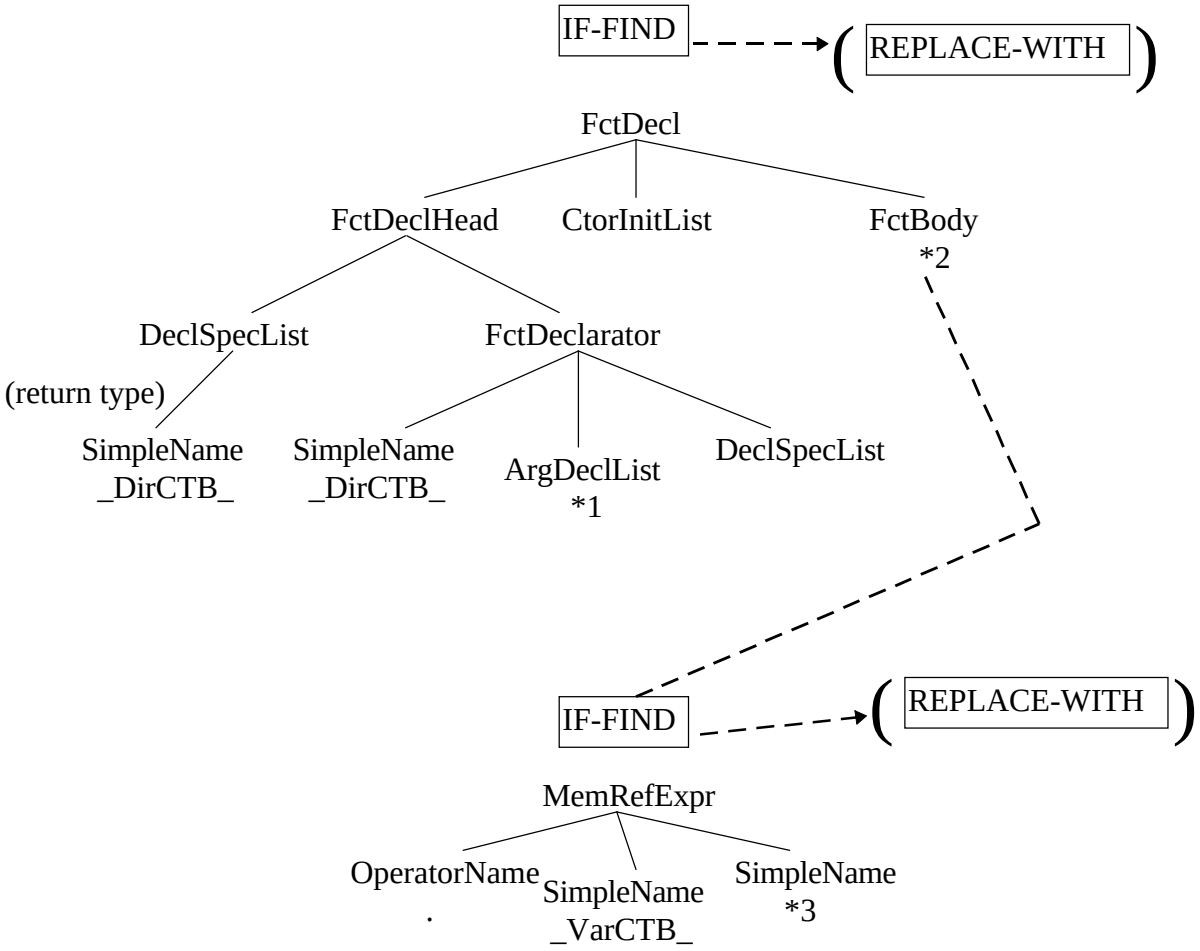



**Figure 5.1. Basic if-find patterns for a function**

The overall use of patterns can perhaps be seen more clearly from this diagram: it is anticipated that there will be several patterns for converting structs, and several for converting MFTBs. There will be a ‘top-level-pattern’, that will store all of these different patterns, and this pattern will be matched against the whole program. For a certain MFTB, only one of the top-level-pattern’s child patterns will match, and this one will be used. For the example in figure 5.1, only functions returning a CTB by value will match, and so the sub-patterns for the function body will be only for conversions required in those circumstances.

Finally, the idea of having sub-patterns needs to be generalised and extended, as they may be needed in patterns other than for function bodies. It also may be necessary to have a sub-pattern within a sub-pattern, for example for an MFTB call from inside the body of another MFTB. This is shown in figure 5.2, and

the desired effect is to keep most of the arguments (sons of the *3 node), but find and remove the CTB one.

This requires any level of sub-pattern to be possible, which is achieved by allowing if-find nodes to optionally have a sub if-find/replace-with attached to it.

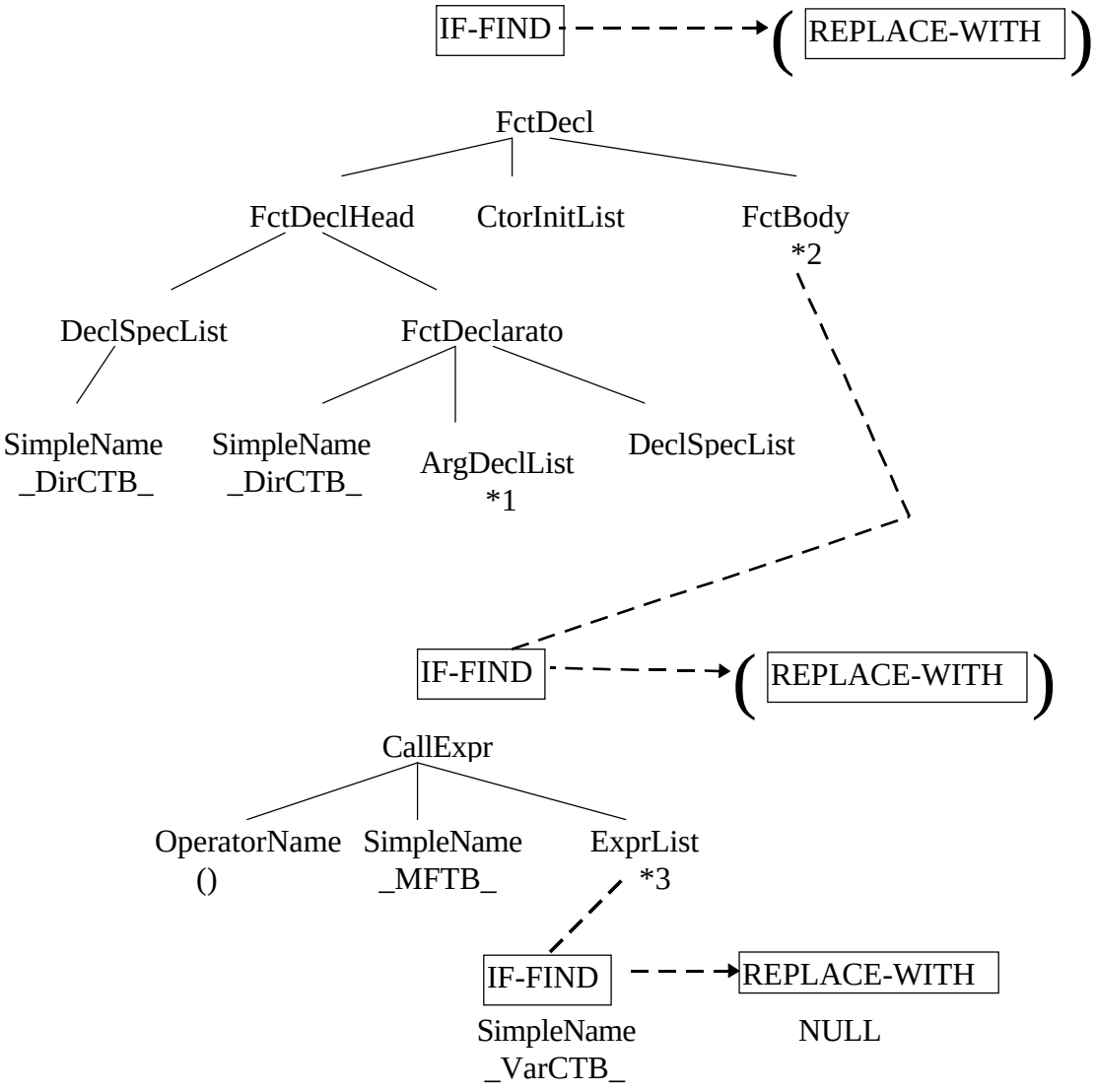



**Figure 5.2. Need for multiple levels of sub-pattern**

### 5.1.6. Summary and example

To recap the pattern semantics so far, a description of the data types explicitly and implicitly created will be given, and an example of how a pattern would apply to a section of code will be shown.

A transformation is specified by the combination of an if-find node and a replace-with node, and this combination will from now on be called an if-replace-spec. The top level transformation is specified by a list of if-replace-specs, one for each type of MFTB and CTB definition. It will also be necessary to have several different patterns to apply within a function body, so the appropriate if-find nodes will also have a sub- list of if-replace-specs. Such a list will be referred to as a trans-spec (transformation specification), or just a ‘pattern’. These data structures are shown in OMT object form in figure 5.3, and the attributes of the two node types are also shown, with optional ones enclosed in square brackets.

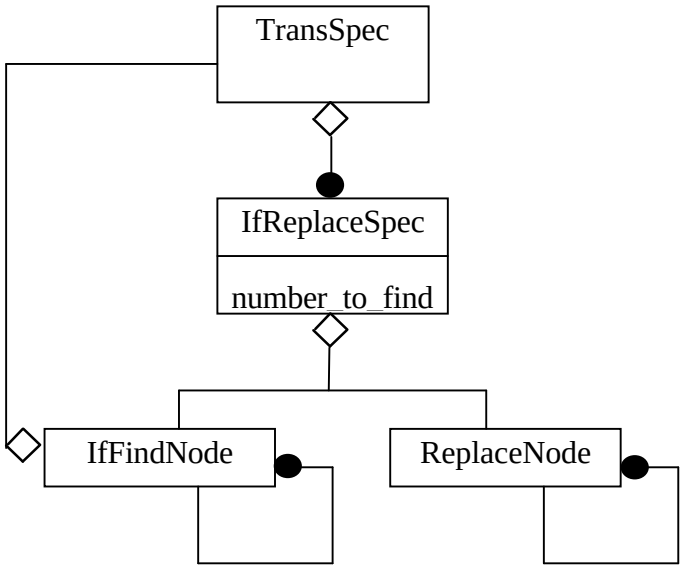



**Figure 5.3. OMT type diagram of pattern components**

An example of the application of a pattern will now be presented, to help clarify the ideas presented above. The example is shown in figure 5.4; the input program is shown in figure 5a), the tree of the input program (note the tree syntax used is exactly that of the Cppp parser) in figure 5.4 b), the if-find-spec applied in figure 5.4 c), and the converted program in figure 5.4 d).

The pattern shown converts an MFTB with a CTB argument passed by reference. It only has one if-replace-spec for the FctBody, which converts data member accesses; a real example would have many (for MFTB calls, CTB declarations, and so on), and would be used together with a pattern to convert the struct.

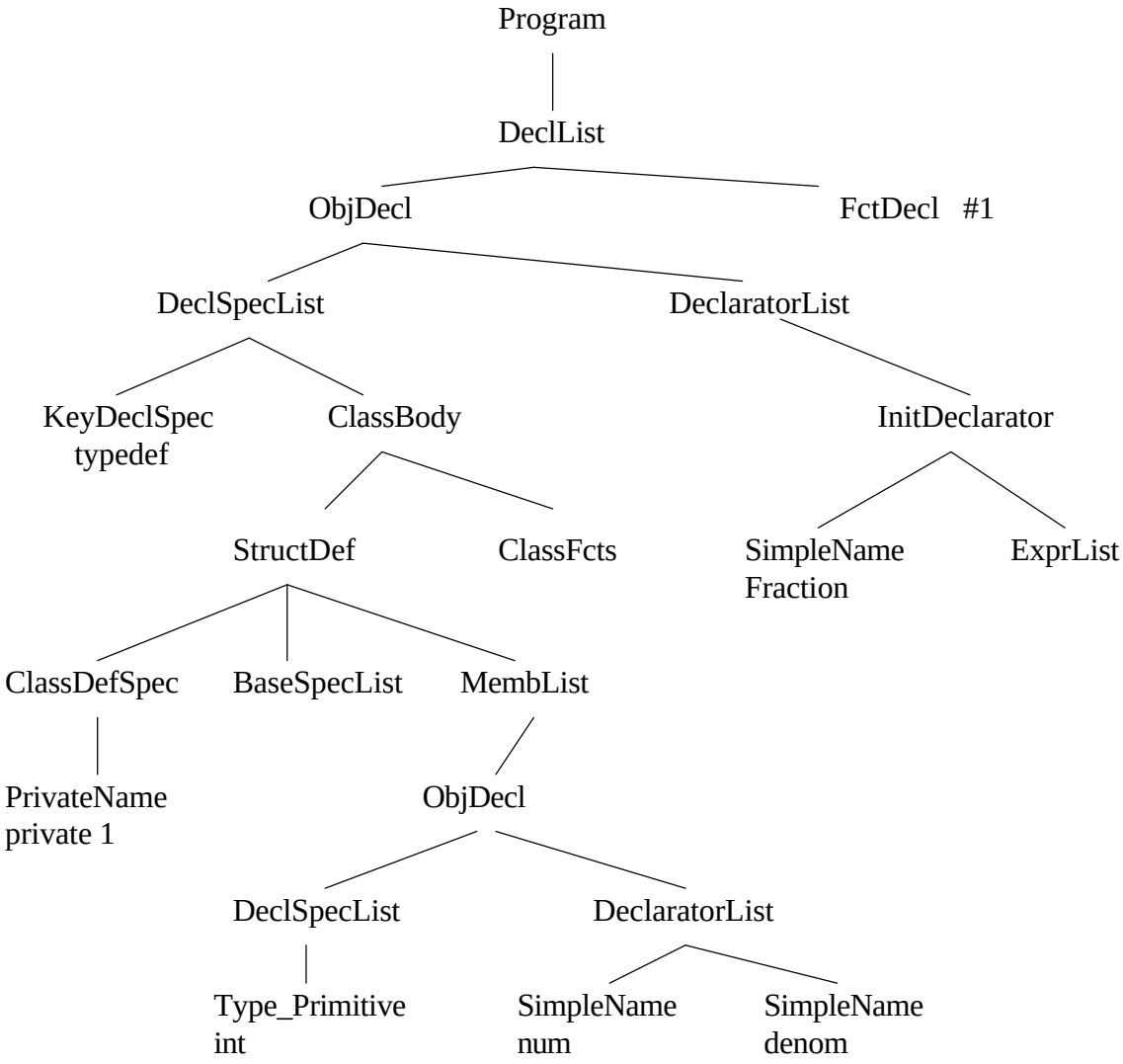
The program is a simple example, containing a struct that represents a fraction, and a function to read two integers into a Fraction variable.

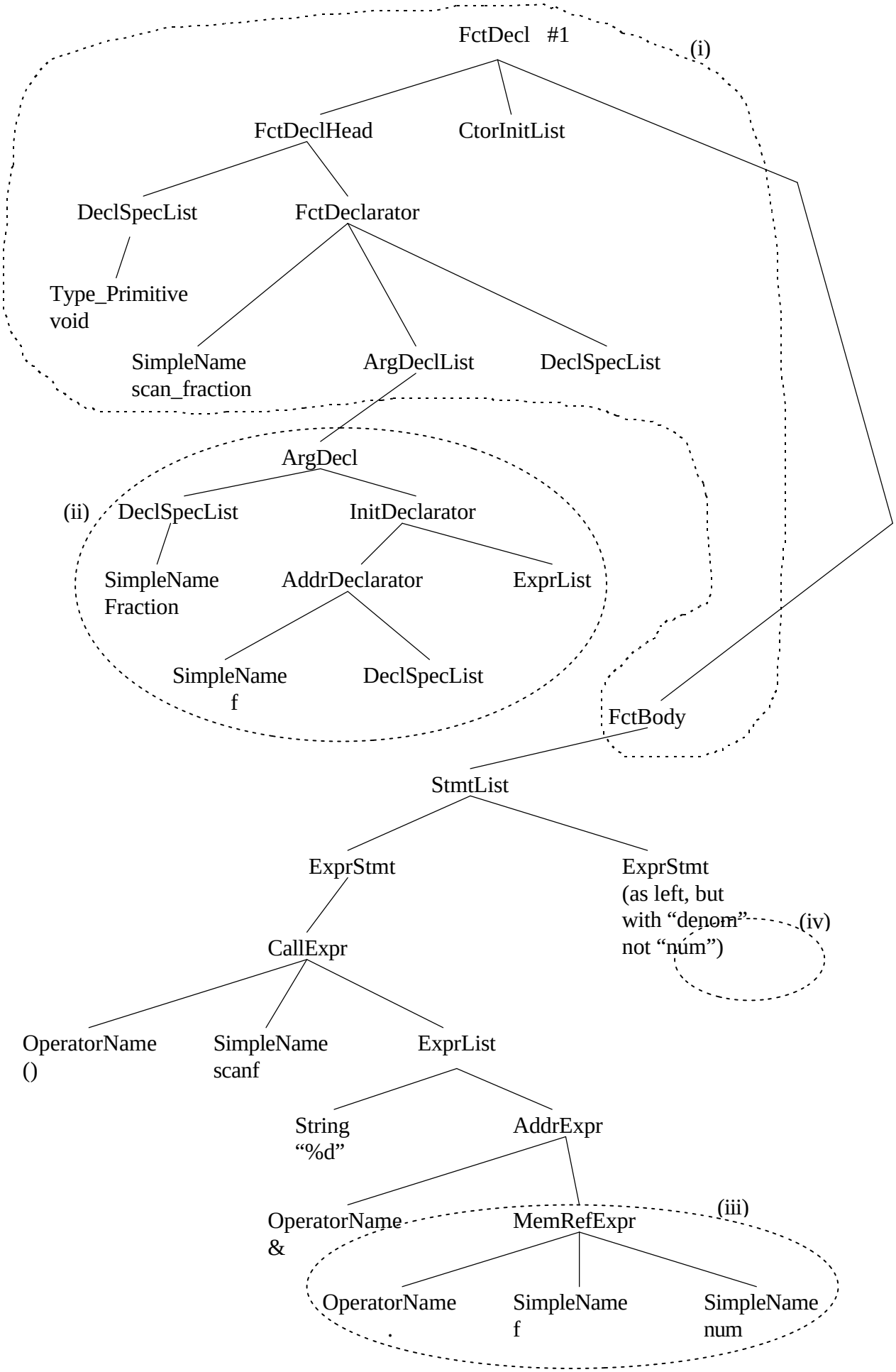
In figure 5.4b), the code sections that are modified by the patterns are encircled, and it can be seen that the ObjDecl node representing the typedef and struct is not changed. The FctDecl section enclosed by (i) matches the if-find pattern labelled (v) in figure 5.4 c). Note that the DeclSpecList and CtorInitList

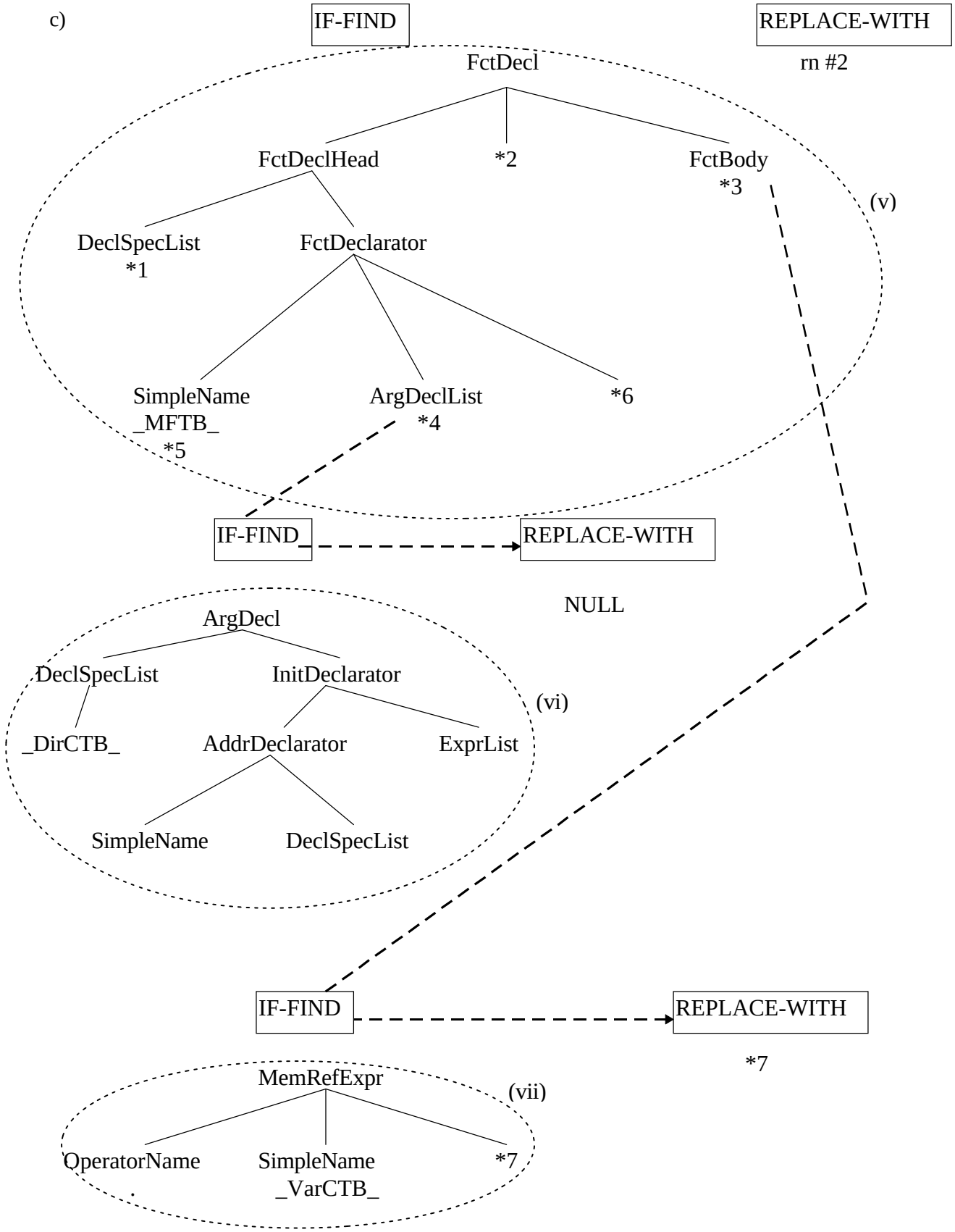
a)

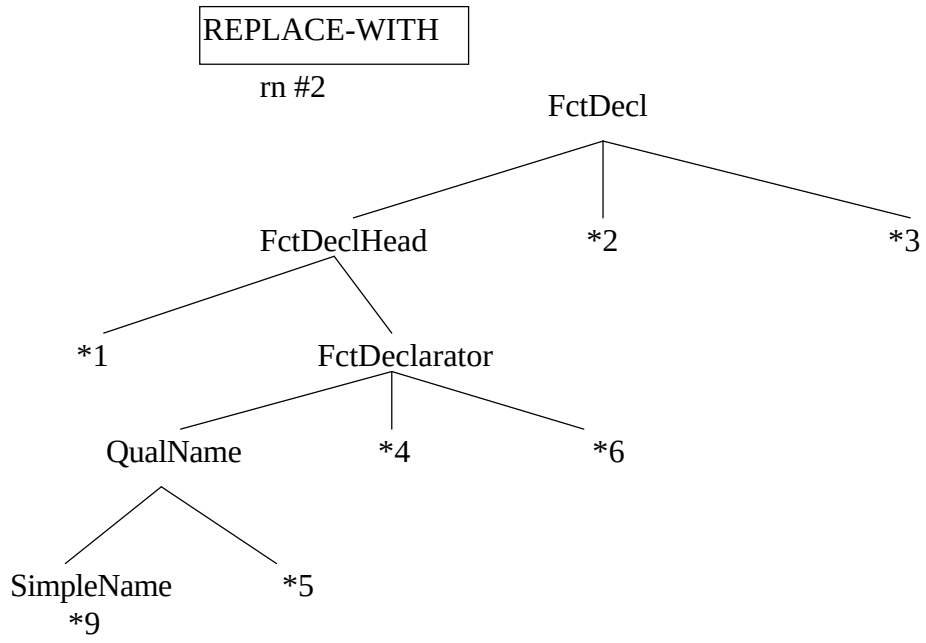
```
typedef  struct {
      int num, denom;
            } Fraction;
void scan_fraction (Fraction & f) {
      scanf(“%d”,&f.num);
      scanf(“%d”,&f.num);
}
```

b)




FctDecl #1
(i)
FctDeclHead
CtorInitList
DeclSpecList
FctDeclarator
Type_Primitive
void
SimpleName
scan_fraction
ArgDeclList
DeclSpecList
ArgDecl
(ii)
DeclSpecList
InitDeclarator
SimpleName
Fraction
AddrDeclarator
ExprList
SimpleName
f
DeclSpecList
FctBody
StmtList
ExprStmt
ExprStmt
(as left, but
with “denom”
not “num”)
(iv)
CallExpr
OperatorName
()
SimpleName
scanf
ExprList
String
“%d”
AddrExpr
OperatorName
&
(iii)
MemRefExpr
OperatorName
.
SimpleName
f
SimpleName
num


c)
IF-FIND
REPLACE-WITH
rn #2
FctDecl
FctDeclHead
*2
FctBody
*3
(v)
DeclSpecList
*1
FctDeclarator
SimpleName
_MFTB_
*5
ArgDeclList
*4
*6
IF-FIND_
REPLACE-WITH
NULL
ArgDecl
DeclSpecList
InitDeclarator
(vi)
_DirCTB_
AddrDeclarator
ExprList
SimpleName
DeclSpecList
IF-FIND
REPLACE-WITH
*7
MemRefExpr
(vii)
OperatorName
.
SimpleName
_VarCTB_
*7




d)

```
class Fraction {
      int num, denom;
};

void Fraction::scan_fraction () {
      scanf(“%d”,&num);
      scanf(“%d”,&num);
}
```

**Figure 5.4. Complete example application of a pattern**

nodes in the pattern have star numbers - whatever the corresponding sections in the syntax tree are, they are copied directly to the same star number in the converted tree.

The SimpleName (node type for identifiers) with the FctDeclarator parent represents the name of the function. In the if-find pattern, this has a special match indicating that it must be an MFTB - the star number is only used here so that the name can be stored and used in the new tree, where it is made into a qualified name of the form "*ClassName*::*FuncName*".

The ArgDeclList has a star number and a subsidiary trans-spec, which means that when this star number is incorporated into the new syntax tree (via the replace-with pattern), it will have been modified according to the subsidiary trans-spec. It can be seen that the if-replace-spec (vi) belonging to the trans-spec matches the argument declaration in the syntax tree (ii). The DeclSpecList describes the type of the declaration - the special match means this must be a CTB argument, and it is completely removed for the new member function.

The FctBody also has a star number and a subsidiary trans-spec; therefore it is modified before being transferred to the new tree. The modification

here is to replace statements of the form "StructName.member" with "member" (shown in section (vii) of the if-find pattern). Here there are two places in the input where a sub-tree matching this occurs ((iii) and (iv)), and both of them will be converted. It does not matter that the statements are embedded in a AddrExpr, which is embedded in a ExprList - they will still be replaced.

The code for the transformed version is given in figure 5.4 d), where it is assumed that the struct has been converted by another pattern.

## *5.2. Extensions*

The basic ideas have now been covered, but there are several more subtle reasons why the pattern matching language is not expressive enough. Many of these can be seen in the example presented above in figure 5.4, and the required extensions are described below.

### 5.2.1. Number to find

For an if-replace-spec, it has been found useful to be able to specify how many matching patterns should be found in the source tree. An example of this can be seen in figure 5.4 c), where the return type of the function is allowed to be anything (by the *1 in the DeclSpecList). However, due to the ambiguities with multiple CTB arguments discussed in section 3.3.5, functions with more that one CTB appearing in the function prototype are not going to be converted. Therefore the return type can be anything except the CTB.

This is enforced by attaching a trans-spec to the DeclSpecList node, with a pattern matching the CTB type-name, and the condition that none of these patterns should be found. If a matching pattern is found, the if-replace-spec will fail, and so will the parent trans-spec (attached to the DeclSpecList). Therefore the DeclSpecList node will not match, and the whole function definition will not be converted. However, the (implicit) trans-spec to which the if-replace-spec for the FctDecl belongs will not fail.

Another similar case is where either zero or one CTB argument must be found in the arguments list for an MFTB.

Note that, in introducing this extension, the concept of a sub-trans-spec has been altered slightly, as it could now influence whether or not the node matches a syntax tree node. There are now several type of trans-spec, firstly a purely conditional trans-spec (C), where the number-to-find for all its if-replace-specs is zero, and therefore the replace-with patterns will never be used. There are also purely modifying trans-specs (M), with no number-to-find for any of its if-replaces, or mixed modifying / conditional (M) (C), with some or all if-replaces having a number-to-find, but not all of these being zero.

The number-to-find is also allowed to be "at least" or "up to" a certain number, for additional flexibility.

### 5.2.2. Comparisons with nodes

The problem here can again be seen in figure 5.4 c), section (vii), where the only check made on the SimpleName whether or not its type is the CTB. This means that if a new variable of the CTB type was declared within the function, it would also be converted which is incorrect.

In fact the check that is needed is that this SimpleName is the same CTB variable as the formal parameter. This is accomplished by using an "IsEqual" special match macro with an "argument", the argument being a node from the input tree identified by a star number.

The need for this was first realised for MFTBs returning a CTB, where the declaration that is removed must be for the same variable that is returned at the end of the function code.

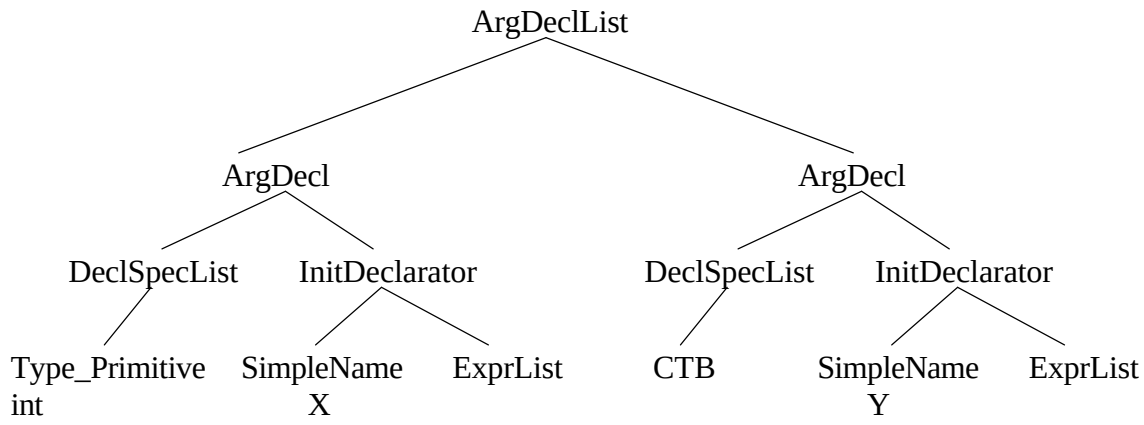
### 5.2.3. Remove son and concat

These extensions are needed because of difficulties with list-type nodes in the syntax tree, which may have any number of sons.

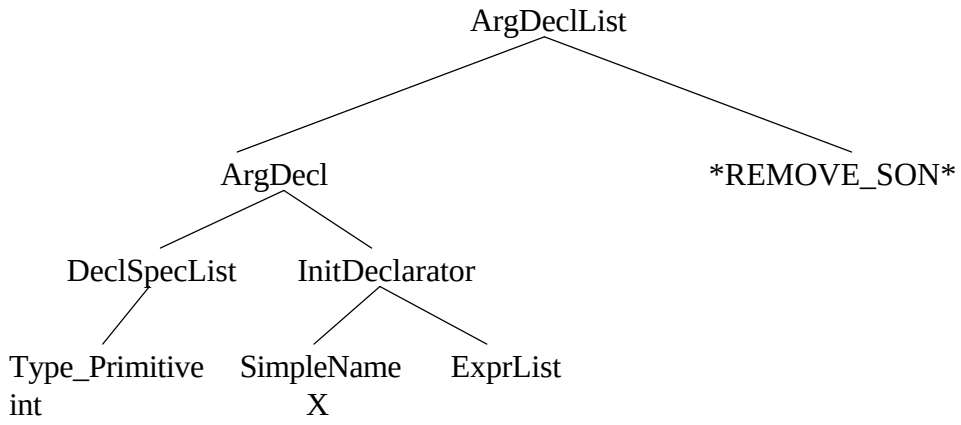
An example is a list of function arguments, where there will often be one argument (the CTB argument) that is to be removed, and the rest must remain. It is not possible to have a pattern with a node for each individual ArgDecl, so how can the replace-with pattern specify which son is to be removed?

In the ArgDeclList replace-with pattern in figure 5.4 c), the argument was replaced by a NULL pointer, which will not actually work. Cppp will not tolerate NULL tree nodes, because it must be able to call member functions on every node. The solution is to have a 'dummy' tree node in the replace-with pattern, with name "*REMOVE_SON*", which will tell the pattern matching engine which node to remove. The stages are shown in figure 5.5, with a) being the original tree node, b) the (temporary) transformed tree node, and c) the equivalent final output tree node.

A very similar problem exists for adding nodes to the existing members of a list-node. This is required for adding protect specifications ("public:", "protected:") to the definition of the new class. These do not exist in structs in C, and need to be added (directly as members) to the MembList node for a class.




Replaced By:




Is exactly equvalent to:

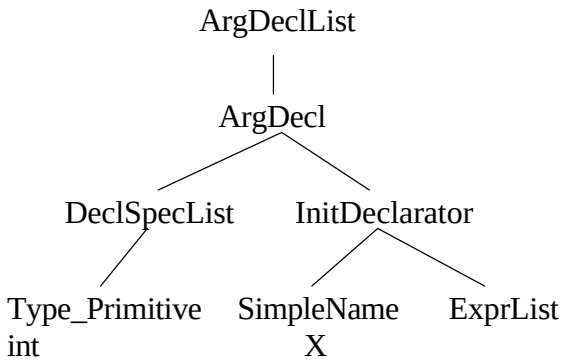



**Figure 5.5. Working of REMOVE_SON**

It is simplest to find the desired protect specification for each variable, and add this protect spec in front of the variable declaration in the MembList. To do this, a dummy node called "*CONCAT*" is used, which means that all sons of the CONCAT node will be added as sons of the CONCAT node's parent. The two equivalent forms of the modified tree are shown in figure 5.6.

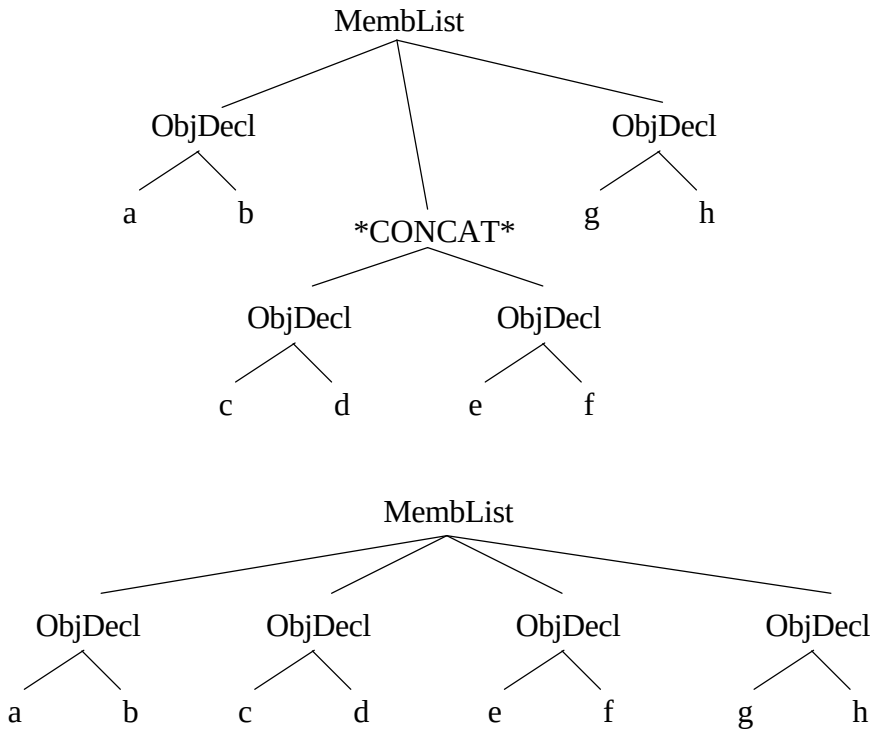



**Figure 5.6. Working of CONCAT**

### 5.2.4. Special creates

In the example in section 5.1, a star number has been used for re-creating the CTB name. This will not generally work, as there are several ways of defining the name of a struct (see section 3.3.4) and any one pattern will not always find the struct name.

Also the name of the new class may be chosen by the user, and so a consistent way of incorporating this name into the modified patterns is required. The extension devised to perform this is essentially a special match system for replace-with nodes, using "special create" macros.

These macros have been used in two cases so far, firstly as described above (_CTBName_), and secondly for producing the prototypes for the MFTBs to include in the class definition (_MFTBHdrs_).

### 5.2.5. Names of nodes

All nodes in the Cppp syntax tree have a label (the ‘name’ of the node), identifying the C++ construct that they represent, but for some nodes in the if-find pattern it is not important what this is. Therefore, the name for an if-find node is allowed to be *ANY*, and if this is the case, any name in the source tree matches.

However, for terminal symbols such as identifiers or type specifiers, information other than the type of the node is recorded. The names of identifiers are specific to an individual program, and should not appear in a pattern, but there are cases when this additional information needs to be checked. An example of this can be seen in figure 5.4 b), where the KeyDeclSpec (a descendent of the ObjDecl node for the struct) also has a "typedef" label, and if this label was not "typedef", the conversion would be different. Also operators such as "->" and "." all appear in the syntax tree as OperatorName nodes, and protection specifications such as "public" and "private" all appear as ProtectSpec nodes.

Therefore, for certain kinds of nodes, an additional name can be specified, both for if-find and replace-with versions.

### 5.2.6. Star nodes with sons

The final extension deals with list-nodes again, this time when new nodes are to be added to a list node, after any sons it already has.

Initially, it may seem that creating a CONCAT node with the old sons and the new sons as children will perform this task, but it is not possible to add the old sons individually to a CONCAT node the only option is that shown in figure 5.7, which is incorrect.

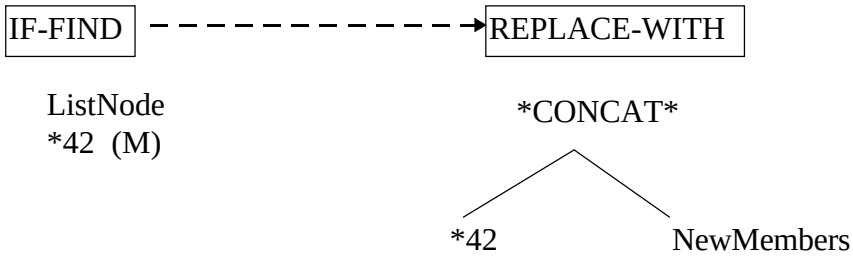



**Figure 5.7. Incorrect use of CONCAT**

As the old sons are to be kept, the original list-node must be labelled with a star number. Previously, any if-find or replace-with node that has a star number cannot have any sons, but this extension is to allow replace-with nodes with a star-number and sons. These sons will be added to the star-node (copied from the original tree) as extra sons.

This requirement arises from the need to add function prototypes to the existing list of members (MembList) for the definition of the created class.

## *5.3. Complete description of pattern language*

### 5.3.1. Application

The essential data structures are almost unchanged by these extensions, with the only difference from the object diagram given in section 5.1.5 figure 5.3 being that the if-replace-spec has the number-to-find as an attribute.

The actual algorithm used to apply a pattern to a syntax tree can now be clarified. The trans-spec is tested on each node of the input tree, using a pre-order traversal. For each node, the if-replace-specs are tested in order, to see if their if-find node matches the syntax tree node. If it does, then the node is replaced by the replace-with nodes of the if-replace-spec, and the traversal skips the sons of the new node. If it does not match, the syntax tree node is left unmodified, and the trans-spec is applied to its sons.

### 5.3.2. Comparison with other pattern matching algorithms

As the final version of the pattern matching concept has now been described, this can be compared with the pattern matching ideas from compiler techniques described in section 9.12 of Aho *et al* [12]. Here pattern matching is used, on a syntax tree that has been modified to represent basic machine operations, to output the assembly code for the input program.

This pattern matching uses if-find and replace-with parts, but it has in addition an action to perform (output the code) when a match is made. The if-find nodes have a basic kind of special macro system, in that the run time value that a node’s attribute must have for it to match can be specified There is no equivalent to the idea used in ctrans patterns of matching one node against a previously found node.

A star-number system is also used, but not explicitly, as the replace-with node is the same node as the one in the original pattern with that type. However, the replace-with pattern is only ever one node, so there can be no equivalent of the ctrans REMOVE_SON and special create features.

The major difference is perhaps that the patterns in [12] can only be applied against the nodes at the very bottom of the syntax tree. This is not the case in ctrans because of the star-node system, which means that nodes near the root of the tree can match a pattern without the pattern having to specify the exact sons this node must have. This also allows embedded patterns to be used, which provide much of the power of the system developed in ctrans. This is because the context in which a certain transformation must occur can be specified by it's parent patterns

(for example, removing the return statement is only performed for function bodies where the parent pattern has found the return type of the function to be the CTB).

This key difference is represented in the algorithms used: both apply patterns during a depth first traversal of the syntax tree, but ctrans uses a pre-order traversal and Aho *et al* use a post-order traversal. This allows patterns to be matched from the top down in ctrans, whereas patterns are matched against the leaf nodes first in the code-generation algorithm.

### 5.3.3. Node components

A brief description of all the possible components for the two types of pattern node is given in this section. In fact, these components can only appear together in certain combinations, but these are shown in the different node types used for the semantic definitions found in sections 5.3.5 and 5.3.6.

If-Find nodes:

- name - a text name, either "*ANY*" or the required name for the input tree node
- star-num - the star number, implying a node automatically matches
- spec-match(arg)- the special match macro, and possibly a star-number argument
- sons - a list of if-find nodes
- trans-spec + ( mod | cond | cond+mod ) - a trans-spec, with a tag to indicate the whether it is modifying, conditional or both.

Replace-With nodes:

*either*

NULL

*or consist of:*

- name - the name the created node is to have
- star-num - identifies a node from the original tree
- spec-create - the special create macro
- sons - a list of replace-with nodes
- REMOVE_SON - identifies a dummy node to be ignored
- CONCAT - identifies a dummy node - add sons to its parent

### 5.3.4. Conventions for descriptions

In the sections below, more formal semantics are presented for the pattern nodes, with if-find nodes covered in section 5.3.5., and replace-with nodes

in section 5.3.6. For the if-finds, the conditions for a syntax tree node (or ast node, for Abstract Syntax Tree) to match are enumerated for nodes with each possible combination of components, and similarly the ast nodes created are described for each possible replace-with node.

The syntax used is a mixture of formal logic, the C programming language, and mathematical notation, with the result being somewhat less than formal, but hopefully also unambiguous.

Constructs used are:

standard C syntax for arrays
++ for array concatenation
$[\ f(X)\ ]_{X=\ Xstart,\ Xfinish}$ to represent a list of f evaluated for all integer X values from $X_{start}$ to $X_{finish}$

An if-find node is represented by the constructor ifn(..), with the arguments being the node name and zero or more optional components. A replace-with node has the constructor rn(..), with at least one of the possible components as an argument. Syntax tree nodes are represented by the constructor astn, which has the node name and an array containing ast nodes that are its sons as arguments.

The predicate function 'match' is used in three forms: firstly with an if-find node and a syntax tree node as arguments, with the result being whether the ast node matches the if-find; secondly with a trans-spec and an ast node as arguments; and thirdly with a special match macro and an ast node as arguments.

The function 'create' produces the syntax tree node that will replace a node that matches an if-find pattern, using the information stored in a replace-with node. The function returns an ast_node, and is used in two forms: firstly with a replace-with node as the argument, and secondly with a special create macro as the argument.

Where the conditions to match are the same for several different if-find node constructors, these conditions have only been written once. Also note that where a specific number of sons have been used for replace-with nodes, this is only for clarity and the semantics apply to any number of sons.

Finally, the issue of storing star-nodes has been avoided, again for clarity. The match function will in fact have the side effect of storing the ast node in a star-list for any if-find node that matches and has a star-number as one of its components. This star-list should be passed as a argument to both the match and create functions, and access to an ast node stored in the star-list is via the functions 'starName(num)' and 'starSons(num)', to retrieve the name and sons of node stored with star number 'num'.

### 5.3.5. If Find nodes

match( ifn(name, star_num), astn(astname,asons[num_asons]))
match( ifn(name, star_num, [mod]trans-spec),
astn(astname,asons[num_asons]))

*iff* (name = ANY) ∨ (name = astname)

---

match( ifn(name, sons[num_sons]), astn(astname,asons[num_asons]))

*iff* [(name = ANY) ∨ (name = astname)] ∧ (num_sons = num_asons)
∧ ∀i: 0 ≤ i < num_sons. (match(sons[i], asons[i]))

---

match( ifn(name, [cond]trans-spec), astn(astname,asons[num_asons]))
match( ifn(name, [cond]trans-spec, star_num),

astn(astname,asons[num_asons]))
match( ifn(name, [cond/mod]trans-spec, star_num),
astn(astname,asons[num_asons]))

*iff* [(name = ANY) ∨ (name = astname)]
∧ match( trans-spec, astn(astname,asons[num_asons]))

---

match( ifn(name, spec_match), astn(astname,asons[num_asons]))
match( ifn(name, spec_match, star_num),
astn(astname,asons[num_asons]))

*iff* [(name = ANY) ∨ (name = astname)]
∧ match( spec_match, astn(astname,asons[num_asons]))

### 5.3.6. Replace nodes

create( rn(star_num) )

= astn(starName(star_num), starSons(star_num))

---

create( rn(star_num, sons[num_sons]) )

= astn(starName(star_num), (starSons(star_num) ++ sons) )

---

create( rn(spec_create) )

= create( spec_create )

---

create( rn(name, {rn1, rn2, rn(REMOVE_SON), rn4}) )

= astn(name, {create(rn1), create(rn2), create(rn4)} )

---

create( rn(name, {rn1, rn2, rn(CONCAT, {rnA,rnB,rnC}), rn4}) )

= astn(name, {create(rn1), create(rn2), create(rnA),
create(rnB), create(rnC), create(rn4)} )

---

create( rn(name) )

= astn(name, NULL)

---

create( rn(name, sons[num_sons]) )

= astn(name, { [ create(sons[i]) ]$_{i=0, (num\_sons-1)}$ } )

## *5.4. Actual Patterns*

### 5.4.1. Creating

While a basic idea for several patterns had been formed from the analysis performed on the syntax trees produced by C-tree, these patterns needed many refinements and extensions, and also converting to Cppp tree syntax.

Therefore, before the design work began in earnest on the pattern matching engine and the rest of the tool, many example programs were created, and hard copies were produced of the equivalent Cppp syntax trees. For some of these, a graphical 'tree' representation was also created (by hand), so that they appeared in a similar form to the pattern examples shown above.

These sample trees were vital in the development not only of the actual patterns, but also of the semantics for applying patterns.

The creation of patterns was in fact a tricky and time consuming task, with most patterns gradually being refined to their final form throughout the coding and testing of ctrans.

For each pattern, the exact effect of using nodes with certain features has to be understood exactly to achieve the correct transformation. It is easy to misunderstand the conversion that a pattern represents. Examples of where a pattern is not immediately obvious are the pattern shown in figure 5.7, which replaces a list-node with the list node and the new members, or an if-find node with just a name, which actually specifies that the equivalent syntax tree node must not have any sons.

Also, whilst creating patterns to convert a certain section of tree as found in one of the example trees, attempts were made to envisage alternative possibilities for the layout of this section. Examples of the issues considered are:

What if node X had sons? What would they be?
What if SimpleName Y had an AddrExpr node as its parent?
What if SimpleName Z was a pointer variable?
Is the conversion affected by this section of the tree?

### 5.4.2. List of basic patterns

All of the patterns used are reproduced in full in appendix B, but they are described briefly below, labelled with the same numbers as in the appendix.

1. CTB Struct and typedef combined
   This will be converted into a class
2. CTB Struct only
   Also converted into a class
3. typedef struct *CTB_tag CTB_name*
   This statement will be removed, as a C++ class automatically represents a type
4. typedef struct *CTB_tag *CTB_name*
   Here the reference to struct *CTB_tag* is changed
5. ordinary function
6. Not an MFTB
   Calls to MFTBs are changed as are accesses to CTB member variables
7. MFTB, CTB arg passed by &
8. MFTB, CTB arg passed by *
9. MFTB, CTB arg passed by value
10. MFTB, CTB return by *
11. MFTB, CTB return by value

Patterns not included (due to time constraints) are:

- Adding new member functions to a class
- CTB struct and typedef for CTB *
- MFTB, CTB return &
- MFTB, multiple CTB arguments

### 5.4.3. Full example

A complete example will now be presented, for the combined typedef and struct definition (which is a slightly smaller pattern than the MFTB patterns), and this is shown in appendix A1.

This is to convert code of the form:

```
typedef struct stack_tag {
    Node_type  * top;
}    stack_type;
```

In the if-find pattern, it can be seen that the syntax tree treats the keyword "typedef" and the struct definition as the type specification, and the identifier "stack_type" (in the example above) as a declarator. The star_node *2 is a BaseSpecList, which is used only for classes in C++ to specify any base classes they are derived from.

The struct tag is used to identify the struct as the CTB, with the _CTBtag_ special match. If the struct does not have a tag, this will still work as the parser will create a unique PrivateName node to go here (made unique by an integer in the name).

The replace-with pattern for this top-level if-replace-spec is fairly straightforward, with the nodes relating to the typedef being removed. The only other point of note is that the *3 MembList node has the special create for the MFTB headers as a son, which means that this node for the new function prototypes will be added to the end of the list. The node created by MFTBHdrs will itself be a CONCAT node to allow for more than one MFTB.

The chief purpose of the trans-spec attached to the MembList is to add in ProtectSpec nodes for the struct variables, but it also has a trans-spec on the DeclSpecList for another reason. This has one if-replace-spec (#3), which modifies declarations of the form "struct s_tag *next;", as the struct will not exist in the converted program, to "CTBName *next".

The DeclaratorList of if-replace-spec #2 is written the way it is, with a conditional trans-spec, to allow for more than one declarator (e.g. "int a,b,c;"). The number-to-find of if-replace-specs #4,#5 and #6 specify that all of these declarators must have public protect specification in the new class. Therefore, the CONCAT dummy node is used to insert a ProtectSpec "public" node in front of the declaration.

There will be two more if-replace-specs very similar to #2 in this trans-spec, for finding declarations of all private and all protected variables.

Finally, there is a "default" if-replace-spec, for is the others fail as there are declarations where variables with different protect specification are declared together. Ideally, the declaration should be split up into one for each type of ProtectSpec required, but as this seemed quite tricky, all of the variables will be set to have the most open protect specification (i.e. "public"), as this can not cause any errors in the new program.

# 6. IMPLEMENTATION

## *6.1. Functionality*

The tool produced as a result of this project is called ctrans. It runs under Unix (Sun OS 4.1.3) and is written in C++(GNU g++ v.2.6.3). Unix was chosen for the development environment mostly because of the compiler tools (Lex and Yacc) and parsing packages available, but also because it is good for program development, providing tools such as xemacs and grep. C++ was chosen because it is an object oriented language and because it seemed an obvious choice for a program that reads in C++ code.

The user interface is a command line interpreter, mostly due to having insufficient development time for a graphical interface, but also to enable it to run on any Unix facility. The user commands available are: read in a C/C++ file, display the syntax tree, convert (by selecting the CTB and MFTBs from lists), save the program as code, and display the program as code.

The tool only performs Type I conversions (from a struct - as described in section 3.3), again due to time constraints. However, the pattern matching engine is almost fully developed and has been tested.

The other chief limitation is that the required user interface feature for setting the protection specification of member variables have not been developed, although features in the pattern matching engine do allow this. Additionally, the ability to check where member variables of the CTB are accessed from does not exist, so the tool in fact always converts functions as "public:" and variables as "private:".

The patterns that have been incorporated into the tool are the minimum set of patterns needed to provide a comprehensive range of transformations, as listed in section 5.4.

Ctrans is capable of outputting the converted tree as C++, but this is not quite perfect - there will often be a couple of small syntax errors in the new program.

## *6.2. Ctrans Structure*

The basic dataflows of the final system are shown in figure 6.1. This diagram is presented now to provide an overview and background for the rest of this section, and will be explained more fully later.

Central to this diagram is the tree manager object, which obtains from the user interface and stores information about the required transformation.

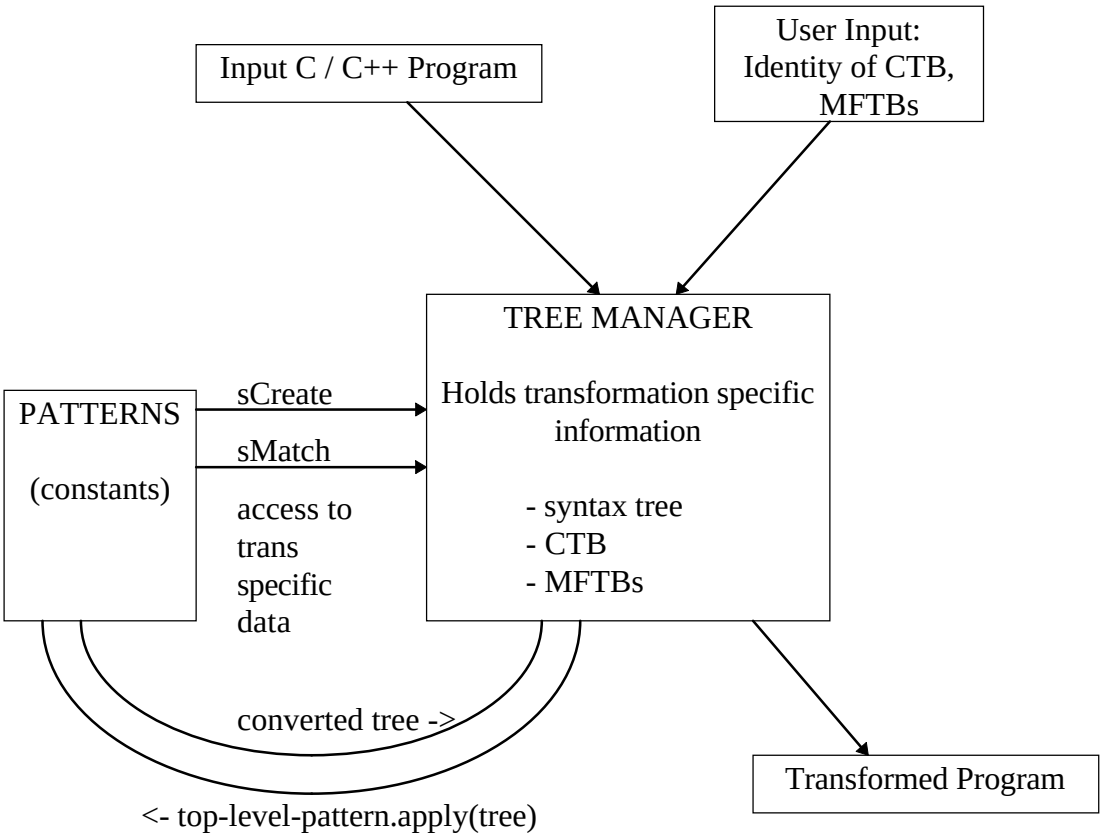



**Figure 6.1. Ctrans data management**

As the run-time information will all be stored by the tree manager, patterns can be declared as constant data items, which is an elegant layout that separates them logically from the actual program.

An input file first has parsing and semantic analysis performed upon it by cppp. The user interface then asks the user for the identity of the CTB and MFTBs, and passes these to the tree manager. To convert the syntax tree, the tree manager calls a member function of the top level pattern for a program. The pattern member functions access global data via only the special matches and special creates, using the sMatch and sCreate member functions of the tree manager.

## *6.3. Cppp*

### 6.3.1. Compilation

Compiling Cppp was in fact a major part of the work done on the implementation of ctrans, and so a summary of the key problems encountered will be given. A fair number of small difficulties were overcome, but there were two

especially time consuming barriers to getting the application to work, and these will be described.

Cppp is a large package, and although it was developed in a different environment from ctrans, a very similar environment is listed by the maintainers as a 'Supported Configuration' [17]. The differences are that ctrans was developed with a slightly later version of the g++ compiler, and Sun-OS4 instead of Sun Solaris. Indeed there are sections of the code in Cppp that take advantage of the fact if the g++ compiler is being used, and before attempting the task, it was felt that it would be straightforward to get Cppp to compile.

The first major difficulty was an error reported by the compiler as an "ambiguous call of overloaded function". This occurred in a string utility class, but the definition for this class and many of its functions are produced by calling a pre-processor macro. This made it hard to follow the code, and also it was suspected for some time that the problem may be down to incorrectly invoking the cpp pre-processor. This suspicion was reinforced by the fact that their had been some initial problems with locating a suitable pre-processor on the system, but the problem turned out to be completely separate. A lot of time was also spent adjusting the compiler options, and compilation parameters and configuration variables in the makefile, but eventually the problem had to be fixed by a modification to the source code.

The second problem was a large list of undefined variables produced by the linker, for which there seemed to be no explanation. The reason for this was eventually found to be that a g++ specific '#pragma' command had been used, which enabled definitions to be placed in header files, and only be actually defined in one of the files '#include'ing the header. Unfortunately, the convention for using this had been changed between the version of g++ that Cppp was tested with and the version that ctrans was developed with.

The bugs remaining after this were fixed relatively easily, but a few more days were spent learning the basics of Cppp data structures and the file structure of the package.

### 6.3.2. Structure

Cppp is based around an object-oriented abstract syntax tree (this is an abstract tree because it has fewer unnecessary nodes, representing intermediate constructs, than a parse tree as specified by a Yacc grammar). There are four main phases in the operation of the Cppp application, starting with the basic parsing phase that builds a syntax tree for the input program. Then, if not suppressed by the user, there is a semantic analysis phase that builds type and scope information into the tree. If specified by the user, the nodes of the tree are then printed out, and finally a user-defined tree traversal can be invoked. Firstly an overview of the tree node classes will be given, and then these phases will be described in order.

The base class for the abstract syntax tree is CpppAstInfo, and as the main purpose of Cppp is envisaged to be for incorporation into other projects, there are many access functions defined for this class. There is also a 'wrapper' class, CpppTree, that essentially just stores a pointer to a CpppAstInfo. The other name of note is CpppAst, which is actually a typedef name for CpppAstInfo *, and which will be used as a general term for all derived classes of CpppAstInfo.

CpppAstInfo stores an array of CpppAst sons, and the start and end locations in the input file of the declaration that the node represents. It also stores the type or 'name' of the node, and several boolean tags about the state of the node. Many of the access functions are for the sons, for example numSons(), removeSon(int) and son(int), and there is also an iterator class (CpppAstListIter) to iterate over the array. The other functions of note are astName(), which returns the (text) type of the node, and copy() which returns a duplicate of the node.

The hierarchy of node classes is very large, being up to six classes deep, and is shown in full in appendix C. Several important node classes are CpppAst_Decl, CpppAst_Declarator, CpppAst_Expr, CpppAst_Statement, and CpppAst_Type. Also of special interest is CpppAst_SimpleName, which is used to represent most identifiers, and has the member function name() to access the name of the identifier.

Now the phases of Cppp will be described, starting with the parsing stage. This uses the tools Lex and Yacc, described in section 4.1.2, with the addition of a special lookahead function. The lookahead inserts and manufactures some tokens in the Lex token stream, to disambiguate certain constructs. For example, different tokens are used for a qualified name (using '::') if the name is a class name, member pointer, member variable or an enum type.

The semantic phase follows, and according to the authors, "most of cppp's complexity resides in the functions that do semantic processing" [17]. The semantic processing is initiated by a call to CpppAstInfo::semProcess(), and for each node this calls semPreAction(), semProcess() for its sons, and then semPostAction(). The pre-and post action functions, which are re-defined for all major node types, contain most of the functionality.

This stage also adds 'objects' to the syntax tree, which contain all the semantic information gathered. These objects are all derived from the class CpppObjectInfo, which itself is derived from CpppAstInfo, and they make it unnecessary to have a symbol table or type table.

Next is a user traversal phase, which can only be invoked if Cppp() is called as a library function rather than from the command line. The function then takes, in addition to the normal command line options, a CpppTraverseActions class as an argument. In this class there is a virtual function defined for every CpppAst node type, each node encountered during the traversal will initiate a call to the appropriate function. The user therefore derives a class from CpppTraverseActions, and re-defines functions to examine any node of interest.

Finally the syntax tree is dumped to the standard output. This is via the function CpppAstInfo::print(), which starts a simple traversal. For each node, its name, any extra information such as the value of an integer, and its location in the source file is printed. The sons of each node are printed immediately below it, at a level of indentation one greater.

### 6.3.3. Printing the tree as code

Cppp provided only the node output function described above, and not a function to print the syntax tree out as code again. Therefore such a function was written, and was the most extensive modification made to Cppp. Although it was the last part of ctrans to be coded, it will be described here as it is dependent only on the Cppp syntax tree, and not on the conversions performed by ctrans.

It had been hoped to use the CpppTraverseActions class described in section 6.3.2. to implement this, but for many nodes, it is necessary to perform actions both before and after the nodes sons are visited. An example is a FctBody node, where "{" has to be printed before the sons are, and "}" after them.

A modification to CpppTraverseActions to allow a pre-action and a post-action was considered, but there were nodes that demanded 'mid-actions' also. An example is a BinaryExpr node, where the operator is the first son, and the two operands the second and third sons.

Therefore a member function printCode(ostream, int indent) was defined for (nearly) all CpppAst classes. This was relatively straightforward, but a few issues needed some thought, and a well formatted output was achieved by a process of trial and error.

Some issues that needed consideration were where to put a newline, for example a newline was desired after a class definition, but not after a variable definition, both of which are ObjDecl nodes. Also which node to associate syntactic symbols with, and which nodes to print spaces before or after were considered. A final difficulty was determining at what stage the indentation should be printed.

The resultant output procedure has a few known bugs, a few known omissions, and a few problems due to the limitations of the syntax tree. There are two bugs; firstly, no bracketing is performed on expressions, and secondly, a semicolon is printed after the third 'do-before-next' statement in a for-loop expression. There are several nodes that have been omitted, due to time constraints, for example all template nodes and all nodes dealing with exceptions. Finally the syntax tree is limiting in a few cases, for example the ClassSpec node is used for all of "struct *name*", "enum *name*" and "class *name*". Also C expressions of the form *expr* ? *expr* : *conditional_expr* will be represented in exactly the same way as an if-statement.

## *6.4. Applying patterns*

### 6.4.1. Initial object and method ideas

The start of the design process for the pattern matching engine was the objects, and these were fairly obvious from the pattern format that was required, resembling closely the structure shown in figure 5.3.

A TransSpec object is the top-level object representing a pattern, and consists of a list of IfReplaceSpecs. However, declaring the all the pattern objects as constants imposed certain restrictions on the design. One of these was that lists were found to be too complicated to initialise, as each data item needed a "ListNode" with a pointer to the next data item. It was far easier to use arrays in the pattern objects, which meant that they also had to store the number of items in the array. The other restriction is that all of the member functions for the pattern objects must be ‘const’ and make no modifications to the receiving object, but this did not cause any problems.

The IfReplaceSpec object was also simple, and would contain an IfFindNode, a ReplaceNode, and an integer for the number-to-find. It would possibly have been convenient to also store a "number-found-so-far", but this would need to be updated during execution and would mean that the object could not be a constant.

For the two node objects, however, there were a number of difficulties. The initial idea was to derive both classes from CpppAstInfo, the base node class used by Cppp. The problem with this arises from the fact that the ReplaceNode needs to create a CpppAst node, which will be a "copy of itself". However, a ReplaceNode needs attributes that a CpppAst node will not have, and so cannot create a copy of itself, as it will not be the correct type of node.

Also, a ReplaceNode cannot create a CpppAst node and set the node type attribute of this node to be a certain value, as each node type is a separate class, derived from CpppAstInfo, and there are 153 different node classes. Therefore, a ReplaceNode would need a large and very untidy if-else statement to determine the class of the object that should be created. A much better approach would be to use the createCopy() function already defined for every node class, but this is only possible if the ReplaceNodes could be of the same CpppAstInfo sub-class as the node they have to create.

The second option, which would allow createCopy() to be used, was to add the extra variables required by pattern nodes to the root class, CpppAstInfo. This was rejected because it would mean that the member functions for all the pattern nodes would have to be part of the CpppAstInfo class. Modifying Cppp this much seemed very bad practice, with several potential pitfalls, and was not done.

A better plan seemed to be to create two entirely new classes for the nodes, each of which would have a CpppAst as an attribute. This would be used for comparisons by the IfFindNode, and for duplication by the ReplaceNode, but there was still a problem. This was that the CpppAst node could not be used to specify the sons of the pattern nodes, as the sons of a CpppAst must also be CpppAsts, and the sons of a pattern node must also be pattern nodes.

There were finally two alternatives, the first of which is shown in figure 6.2. Here, the node classes are inherited from CpppAstInfo, and additionally the ReplaceNode node has another CpppAst attribute for duplication. This would be fine, but wastes memory as a CpppAstInfo object has 9 variables that serve as traversal and error markers, two class variables giving the node's location in the source code, and two other variables, none of which are needed by a pattern node.

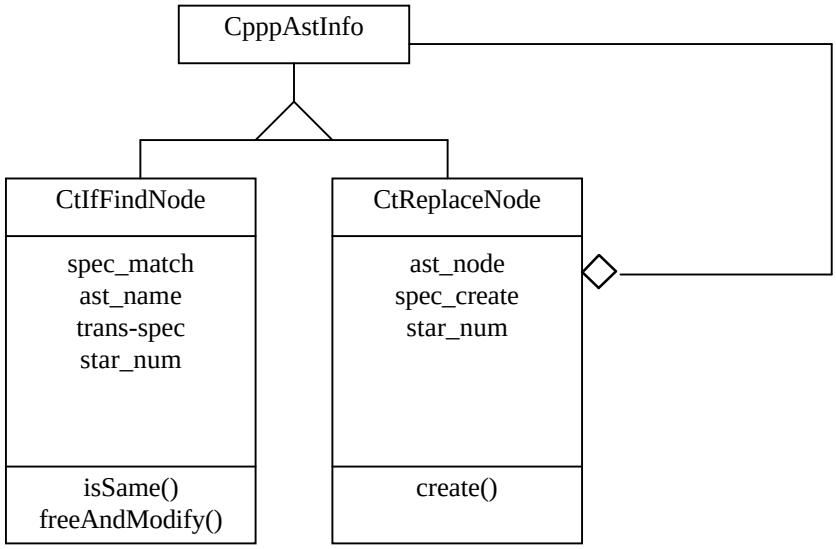



**Figure 6.2. Node design inherited from CpppAstInfo**

The second option, shown in figure 6.3, was to create a new class which would be the base class for pattern nodes. This class would have the star-number, common to both node types, and an array, access procedures, and iterator the sons (as in CpppAstInfo). The ReplaceNode would then have a CpppAst component.

The drawback of this approach is that the functionality provided by CpppAstInfo is duplicated rather than being inherited, and this is not taking advantage of the object-oriented approach. However, it was chosen as it seemed a more complete solution, and there were reasons for trying to keep the memory use of the program down (see section 6.4.3).

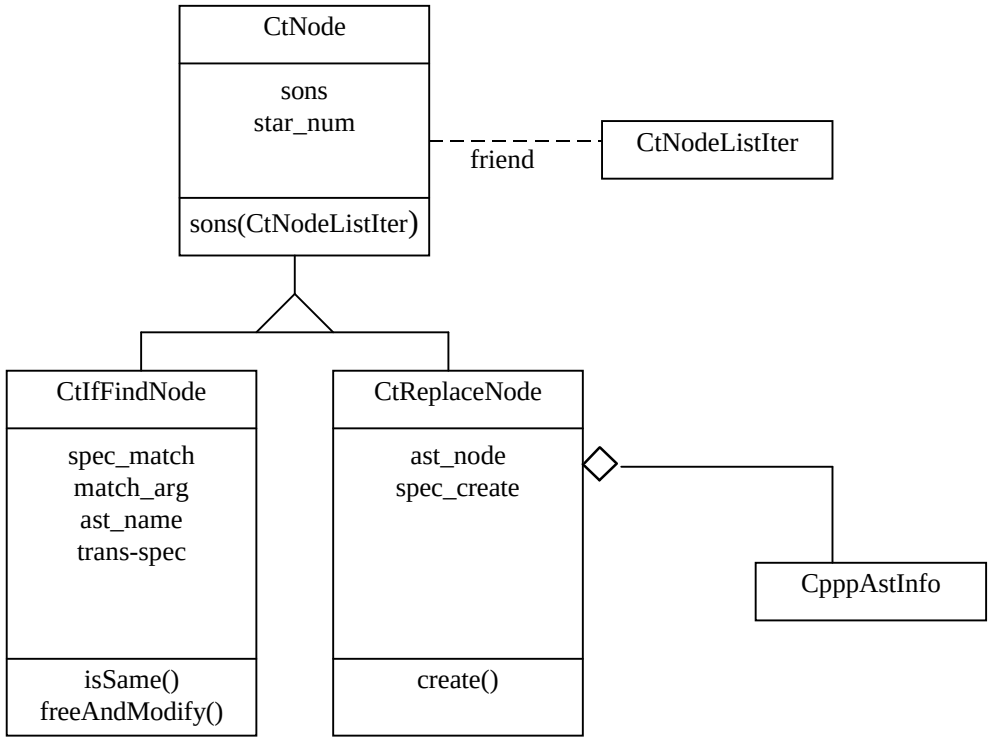



**Figure 6.3. Node design inherited from CtNode**

Having achieved a sound object design, the methods were considered, and these had also become obvious by now. The conversions are performed by three main functions, which are:

```
CpppAst  TransSpec::apply( CpppAst )
Boolean  IfFindNode::isSame( CpppAst )
CpppAst  ReplaceNode::create()
```

These are fairly straightforward, and will be briefly described. Starting with apply(), this iterates through the list of IfReplaces, and for each tests if the if-find node is the same as the input tree (using IfFindNode::isSame). If the result is true, then create() is called for the replace-with node, and this is returned as the result. Otherwise, apply calls itself recursively for the sons of the input node, creating a depth-first traversal of the syntax tree.

The isSame() function essentially contains an if-else statement to find out which components are present in this particular if-find, and performs the appropriate action as described in section 5.3.4. For example, it will apply the trans-spec if the node has one, and will add the input tree node to the star list if there is a star number.

Similarly, create() has to determine the components of the replace-node, and return a the correct node. If the node has a CpppAst syntax tree node, then the result returned is (Cppp node)-> createCopy(), and if there is a special match, then the result returned is the node from the function for creating special create nodes, and similarly for the other cases listed in section 5.3.5.

Also, there is a need for an object that holds global data: firstly the list of star - nodes (star-list) needs to be available within all the functions above. Secondly, the special macros require some information about the current transformation, and this object could hold such information and have the special match and create functions as member functions.

### 6.4.2. Modify or Duplicate?

To discuss the issues raised in this section, a hypothetical syntax tree, using letters(A, B,...) in place of names (FctDecl, WhileStmt,..) has been created, and is shown in figure 6.4. However, the problems discussed also occur with actual patterns.

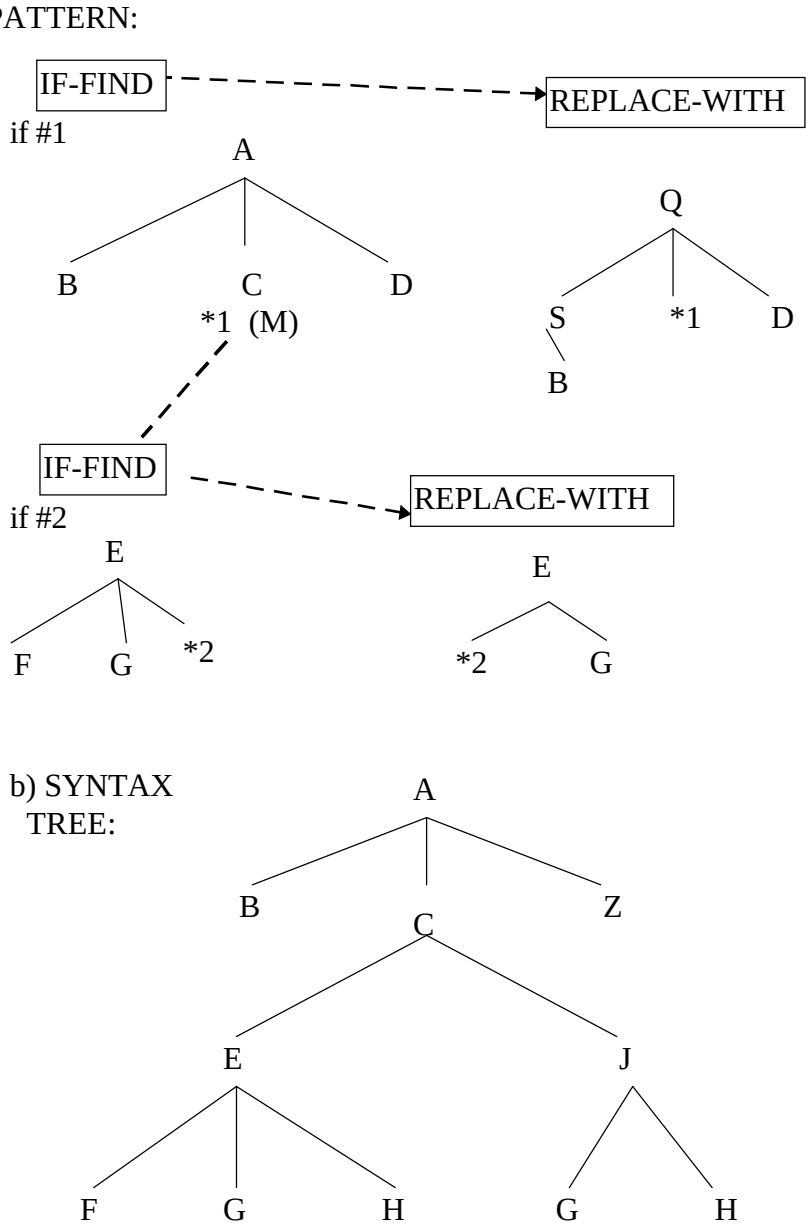



**Figure 6.4. Example to demonstrate modify/duplicate issues**

Firstly, when a section of syntax tree is found that matched the if-find pattern, there is little point in trying to modify this section to make it look like the replace-with pattern. For example, if a section of tree matching the if-find shown in if#1 in figure 6.4 was found, i.e. nodes A,B,C and D, it can be seen that changing these into the corresponding replace-with pattern would be very complicated. Often, replace-with patterns bear even less resemblance to the if-find, and so the best solution is to create all the nodes in the replace-with from scratch.

However, with star nodes it is less clear cut whether the original tree can be used, or it has to be duplicated. This is because of modifying trans-specs, and the case where the pattern in figure 6.4 a) is matched against the tree in figure 6.4 b) will be discussed.

The pattern will be applied to the syntax tree from left to right, and so node C will be checked before node Z. The problem is that it is only found that the pattern does not match when node Z is reached, and at this point the sons of node C have already been modified.

The first solution to this problem was to duplicate the star node, and perform all the modifications on the duplicate, so that the duplicate can be freed if the pattern is later found to not match. For consistency, it was thought best to duplicate even star nodes without trans-specs, and then after the transformation all nodes of the original tree could be freed.

The disadvantage of this approach is that it is not very efficient - even if only one line in a function needs to be changed, the whole function will be duplicated. Also, the issue of checking conditional trans-specs to see if they apply has been ignored.

It was decided that modifying much of the tree would be possible, if it was done in two passes. Firstly, the tree section would be checked to see if the pattern applied - so conditional trans-specs would be tested, but no modifications made to the tree. If the pattern did apply, the tree section would be traversed again and modifications specified by trans-specs would be performed.

Two new functions were created to do this: TransSpec::matches, and IfFindNode::freeAndModify. The latter is called so because it also frees any nodes in the original tree that do not form part of the new tree.

Exactly which nodes would be modified can be indicated using figure 6.4, if we assume that the node Z was a D and so the whole pattern would match the tree.

Firstly, apply() would call isSame for node A, and isSame() would examine nodes A, B, C and D and would return true. Then freeAndModify would be called by apply(), and it would firstly delete node B; it would then call apply() for the sub- trans-spec on node C, which would call isSame() with node C as an argument. This would return false, and so isSame() will be called for node E. It will check nodes E, F, G and H, and return true, so freeAndModify will be called for

node E. This will delete nodes F, G and E, but keep node H. create() will then be called for the sub-replace-with, which will produce nodes E and G, but use the H from the original tree.

The sub-trans-spec will then call isSame() for nodes J, G and H, receiving a negative answer for each of them. Now the freeAndModify call from the first trans-spec will continue, and will delete node D and finally remove node A.

Finally, apply() will call create() for the replace-with pattern, which will create nodes Q, S, B, and D, but get the (modified) node C together with its sons (of which J, G and H are original) from the star-list.

### 6.4.3. Cppp Syntax Trees

When Cppp was chosen to be the parsing package for this project, one of its attractions was the clear syntax tree it produced, which was a faithful and intuitive representation of the input program. Unfortunately, the initial examination of Cppp output was done on syntax trees which had not had the semantic processing stage performed on them, which will be referred to as "parse trees". The "semantic trees" that are the result of calling the semantic processing function on a parse tree are in fact very substantially different, having usually slightly more that twice the number of nodes of the input parse tree.

The key change performed to the tree structure is to replace every variable, function or type declaration with an "object" that represents this declaration uniquely. These objects store information such as the scope, storage requirements, and pointers to (a modified version of) the original declaration, and also every identifier that represents a call to/use of such an object has a pointer to the actual object added as a son.

In fact the semantic tree is slightly more complex that this, as the key top level declarations in the program are (mostly) not replaced as objects, but the objects appear as sons of a "scope" object inserted as a top-level node. While these declarations are more similar to the original tree, they still have significant differences and do not have pointers to their equivalent object (the pointers go the other way), so much of the required semantic information is lost.

It was decided that the transformation could not be based on these semantic trees - there were many problems, but perhaps the key one was the links created between all of the objects and declarations in the tree. These links would have been impossible to produce correctly using a replace-with pattern in any modified sections of the transformed tree, and so it would not have been possible to produce a converted tree in the same format as the original tree.

Mostly, the other problems related to the way semantic trees represented the input program, which made it impossible to reproduce code from the tree that would be the same as the input (while still being semantically correct). For example, if a struct definition was combined with a typedef in the original

code, the two declarations would be separated in the semantic tree. More importantly, although the declaration of a typedef is preserved, the use of it is not. Whenever a typedef name is used in the original program, this is replaced by a pointer to the actual object that the typedef refers to in the semantic tree.

Finally, any implicit casts in the original tree are made explicit, and in fact for almost every use of a variable there is a set of nodes specifying the BuiltInCast call, even if the types appear to match exactly. Also operators are replaced by calls to a BuiltInOperator function, and a definition for each of these required is created. These could perhaps be ignored for outputting the tree as code, but would complicate the if-find patterns enormously.

A large range of alternative approaches was therefore considered, starting with the idea of basing conversions purely on the parse tree. This would mean assuming that all identifiers in the program are unique, which would probably be all right for simple examples, but would not be practical for converting any real program. Therefore it was discarded.

Another idea was to keep two syntax trees, using the parse tree for the conversion, and using the semantic tree to look up any required semantic information. This would not be workable, as there is no way to find the section of the semantic tree that is equivalent to the section of interest in the parse tree.

Therefore, adding links from the parse tree to the semantic tree was investigated. This would be done during the construction of the semantic tree, but the problem found was that the semantic processing functions in Cppp are very hard to follow, and would need to be understood and then have considerable modifications made to them to implement this approach. This problem also made impractical perhaps the most obvious solution, which was to modify the production of the semantic tree so that it is more similar to the parse tree.

The final solution is to duplicate the parse tree, but for certain nodes, add a copy of the memory address of the original node to the duplicate node. The semantic tree is produced from the original parse tree, but the address of many of the nodes are the same as in the original tree. The conversions can now be performed upon the duplicate parse tree, which has pointers from key nodes to the equivalent nodes in the semantic tree.

This is an almost ideal solution, as it was comparatively easy to implement, and allowed access to all the semantic information produced by Cppp. The only drawback is that two syntax trees need to be stored for only one program, which means that ctrans uses more memory than it might have otherwise.

### 6.4.4. Final object design

For the sake of completeness, the diagram presented here (figure 6.5) includes almost all the details of the final object design, although some of these will not be described until later in section 6.

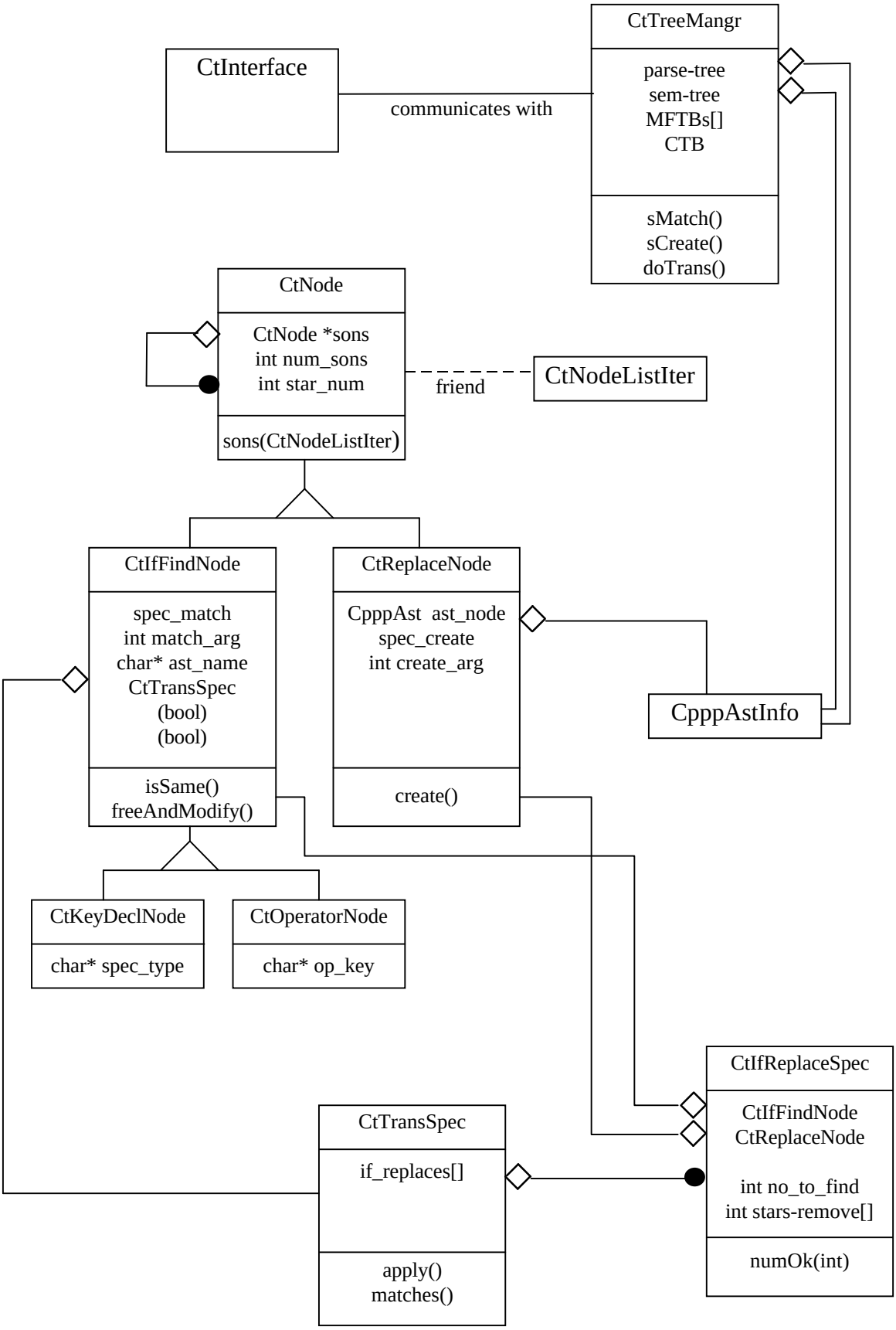



**Figure 6.5. Object diagram for all pattern classes**

### 6.4.5. Final application algorithm

Firstly, ideas for the global information object have been revised, and this object is not going to be passed between the pattern member functions. The only data structure that all the functions need access to is the star-list, and so this is passed as an argument to all five key functions.

The special macros checks and node creation will now be performed by an object that actually is declared globally, called the TreeMangr, for tree-manager. As well as storing information about the CTB and MFTBs, it will store both the parse and semantic syntax trees. It is a global object to enable it to be called from the user interface, to do tasks such as reading in a program file, printing the (parse) tree, and initiating a transformation (which is done by the doTrans() function).

Also, the star-list does have to be effectively global in all pattern member functions, and as they are all recursive, it cannot be declared in any of them. The solution is to declare it in the tree-manager function doTrans(). Additionally, after performing the transformation on the parse tree, this function creates a new copy of the transformed tree which becomes the new parse tree. Semantic analysis is then performed on the transformed tree, checking for errors, and this becomes the new semantic tree.

The other key advancement to the design is the idea of an interface object, which will make calls to the tree-manager methods. The functions of this object are called by a separate command interpreter, and this deals with user input and the command loop.

A pseudo-code summary of the five key functions will now be given. Note that as apply() and matches() must use a very similar algorithm, they both call the function doApplication() with different tags set. This allows an array, to store the number of each if-replace-spec found, to be declared in matches() and be passed as an argument to doApplication. The array will then be the same one when doApplication is recursively called with the son of a Cppp node as an argument, but starts the count of if-replace-specs afresh with each new sub-trans-spec. The total number found can be checked by matches() when doApplication returns, and the responsibility for determining if a number is acceptable rests with a member function of the IfReplaceSpec class.

```
CpppAst CtTransSpec::apply(CpppAst root, CtStarList *stars) {
   int no_found[no_ir_specs];
   CpppAst result;
   const Bool scan_only = 0;

   doApplication(root, stars, no_found, scan_only, result);

   return result;
};


Boolean  TransSpec::matches( CpppAst root, StarList *stars) {
   int no_found[no_ir_specs];
   CpppAst dummy;
```

```
   const Bool scan_only = 1;
   Boolean result = 1;

   for (i=0; i < no_ir_specs; ++i)
      no_found[i] = 0;

   doApplication(root, stars, no_found, scan_only, dummy);

   for (cur_spec = 0; cur_spec < no_ir_specs; ++cur_spec) {
      if (!if_replaces[cur_spec]->numOK(no_found[cur_spec]))
           result = 0;
   }

   return result;
};


void TransSpec::doApplication(CpppAst source, CtStarList *stars,
                        int *no_found, Bool scan_only,
                        CpppAst &result) {
   Bool applied_spec = 0;

   while (!applied_spec && cur_spec < no_ir_specs) {
      spec = if_replaces[cur_spec];
      answer = spec->ifFind()->isSame(source, stars);
      if (answer) {
          applied_spec = 1;
          ++no_found[cur_spec];
          if (!scan_only) {
             spec->ifFind()->freeAndModify(source, stars);
             result = spec->replace()->create(stars);
          }
        }
       ++cur_spec;
   };

   if (!applied_spec) {
      CpppAst son_result;

      result = source;

      for (son_num = 0;
             (son_num < result->numSons());  ++son_num) {

          doApplication( result->son(son_num), stars, no_found,
                                      scan_only, sons_result);
          if (!scan_only)
             result->replaceSon(son_num, son_result);
      };
   };
   return result;
};

Boolean  IfFindNode::isSame( CpppAst test, StarList *stars) {

   if (checkAstName(test))
      if (spec_match != None)
          if (tree_mangr.sMatch(test, spec_match))
              result = True
          else
              result = False;
      else if (cond_trans)
          if (trans_spec->matches(test, stars))
              result = True;
```

```
            else
                result = False;
         else if (star_num == 0)
            result = sonsResult(test, stars);
         else
            result = True;
      else            // checkAstName failed
         result = False;

      if (result == True && !mod_trans && star_num != 0)
         stars->addStar(test, star_num);

      return result;
   }

   void  IfFindNode::freeAndModify( CpppAst test,
                                                  StarList *stars) {
      if (mod_trans)
         stars->addStar( trans_spec->apply(test,stars), star_num );
      else if (star_num != 0)
         /* do nothing - need to keep star nodes,  */
         /* and already added to list in isSame.   */;
      else  {
              CtNodeListIter cnli(this);
              CpppAstListIter cali(test);

              while (cnli) {
                 (*cnli)->freeAndModify( (*cali), stars );
                 ++cnli; ++cali;
              };

              test->freeNode();       // deletes sons array and this
      };
   }


   CpppAst ReplaceNode::create(StarList *stars) const {
      CtNodeListIter cnli;

      if (spec_create != Nothing)
              result = tree_mangr.sCreate(spec_create);
      else {
         if (star_num == 0) {
             result = ast_node->copy(0); }
         else {
             result = stars->getStar(star_num);
         }

         // usually, will be no sons if is star - but otherwise,
         // add the sons after the existing sons in the star node

         for (sons(cnli); cnli; ++cnli) {
             son_result = (*cnli)->create(stars);
             result->addSon(son_result);
         }
      }
      return result;
   }
```

The final issue is that, as described in section 5.2.5, there are circumstances when more than just the type of node (SimpleName, FctDecl, etc) needs to be known. This is implemented by deriving two new classes from IfFindNode, as shown in figure 6.5, for finding KeyDeclSpec and OperatorName

nodes. The checkAstName() function used in IfFindNode::isSame() above normally just compares the ast_name attribute with the name of the Cppp node, but this function is declared as virtual. For the KeyDeclSpec and OperatorName nodes, it is redefined to also check the secondary name attribute of the two nodes, if their first names match.

## *6.5. Maintenance information and utility code*

This section is intended to aid a familiarisation with the C++ code produced by the project. Firstly a brief description of each file in the project will be given, then for some of the small utility files, their contents will be described here, as they are not important to the main algorithms. Finally the conventions used in the code will be described.

### 6.5.1. All files

- ct_local.h - common functions and definitions for all files
- ct_list.h - generic list class
- ct_node.h - contains definitions for the following classes: StarNode, StarList, Node, NodeListIter, IfFindNode (plus derived classes), ReplaceNode
- ct_transforms.h - contains definitions for the following classes: IfReplaceSpec, TransSpec
- ct_manager.h - definitions for the TreeMangr class, plus MemVar and StructClass for storing information on functions and variables, and IntroActions class for finding these
- ct_interface.h - definition for the Interface class
- ct_patterns.h - declarations for all patterns needing to be accessed externally
- ctlist.cc - generic list class
- ctstar.cc - functions for StarList class
- ctnode.cc - contains member functions for the following classes: Node, NodeListIter, IfFindNode (plus derived classes), ReplaceWithNode
- ctifreplace.cc - functions for IfReplaceSpec class
- cttransspec.cc - functions for TransSpec class
- ctintroactions.cc - functions for IntroActions class
- ctmanager.cc - functions for TreeMangr class
- ctinterface.cc - functions for Interface class
- ctmain.cc - main() and command interpreter
- ctastpatterns.cc - Cppp ast nodes
- ctpattern.cc - top level pattern
- ctwords.cc - string handling routines
- cttemplates.cc - definitions for template classes
- pt_output.h
- ptoutput.cc
- ptmain.cc - files for the Pattern Transformer, to automate some of the creation of patterns

- [plus 12 pattern .cc files]

### 6.5.2. Storage of patterns

Patterns are defined as const objects, using the constructor for the appropriate pattern class. Each object has a unique number, and a prefix which is "Ts_" for TransSpecs, "Irs_" for IfReplaceSpecs, "Ifn_" for an IfFindNode or derived class, "Cp" for an object of a derived class of CpppAstInfo, and "Rn_" for a ReplaceNode.

Arrays are defined separately from the object they belong to, and are initialised with the {..} notation. The prefixes used are, "ArIrs_" for an array of pointers to IfReplaceSpecs, "ArIfn_" for an array of pointers to IfFindNode, "ArRn_" for an array of pointers to CtReplaceNode and finally "ArStr_" for an array of int.

The file ctpattern.cc contains the top level trans-spec, which will be applied to the program, and the associated array of if-replace-specs. Each of these if-replace-specs is defined in its own file, and corresponds to one of the ten main patterns listed in section 5.4.2, as listed below.

1. typedef_struct.cc
2. struct_only.cc
3. ctb_typedef_dir.cc
4. ctb_typedef_ptr.cc
5. non_mftb.cc
6. mftb_arg_addr.cc
7.a) mftb_arg_ptr1.cc
7.b) mftb_arg_ptr2.cc
8. mftb_arg_val.cc
9.a) mftb_ret_ptr1.cc
9.b) mftb_ret_ptr2.cc
10. mftb_ret_val.cc

There is also a file called ctastpatterns.cc, which defines constant nodes for all of the CpppAst nodes that may need to be incorporated into a ReplaceNode, a header file ct_patterns.h, and a file func_hdrs.cc that contains special patterns used to produce function prototypes.

### 6.5.3. Lists and star-lists

Although arrays are used for storing lists of items in patterns, lists are still needed to keep track of MFTBs, member variables and similar information, as well as the star nodes. Therefore a generic list template class was created, with a ListItem and List class, as shown in figure 6.6. Note that this has an iterator component built in for convenience.

Also shown in this figure is the star-list class, with extra functions and consisting of StarNodes, which contain a CpppAst and the star-number of that node.

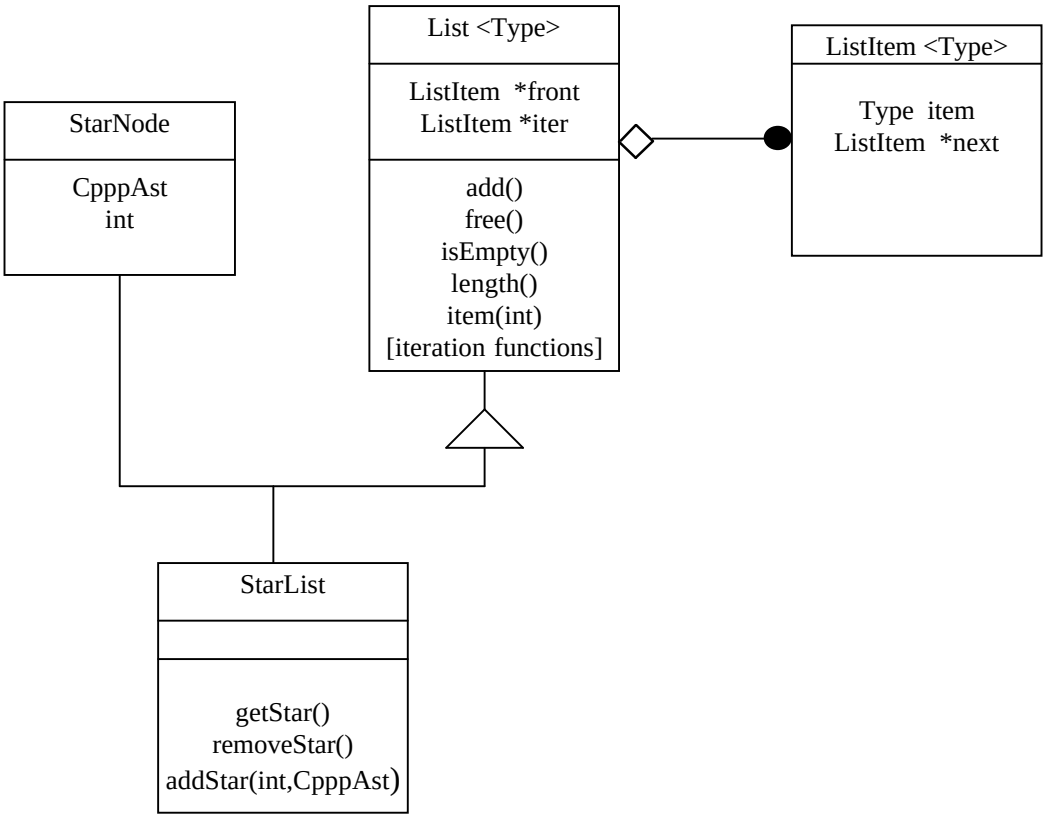



**Figure 6.6. Lists used in ctrans**

### 6.5.4. ct_local

This is a file that is included in all other ctrans files, which has declarations for miscellaneous items and "#includes". A Bool type is declared as the same as int, and an error function, which prints a message and exits the program, is declared inline. Also the Create and Match enumerations for the special macro types is declared here.

### 6.5.5. ctwords

This file contains string handling routines, and was originally written for the pt pattern translator, but is also used in the command interpreter.

There is a function to convert a string into an integer, and a function to split a string into a sequence of words, separated by spaces, and allocate memory for each word.

### 6.5.6. cttemplates

This was required because the version of the g++ compiler being used did not perform intelligent instantiation of templates. This meant that if the template class function definitions file was compiled, no class instances would be generated, and if this file was "#included" in all files using the template class, an instance would be generated for every file thus creating a "multiply defined" error.

The solution was to not compile ctlist.cc, but instead include it in the file cttemplates.cc, and defined all the template list classes used within this file. All other files using template lists would just include the ct_list.h file.

### 6.5.7. Conventions

The conventions used are largely the same as in Cppp, using the project identifier ct, so header files are prefixed "ct_", and .cc files are prefixed "ct".

All types declared in ctrans are prefixed with Ct, so for example the class TransSpec is actually called CtTransSpec.

Variable identifiers are all in lower case, classes and types start each word with an upper case letter, as do constants in ctrans (all in upper case in Cppp). Functions start with a lower case letter, but subsequent words start with a capital, for example freeAndModify().

## *6.6. Tree Manager*

### 6.6.1. Introduction

The tree manager object has two purposes: to provide services for the user interface, and to provide the pattern matching engine with information about the program being converted, via the special macros. The user interface side is covered first, with the information required by the macro functions covered in section 6.6.3, and the functions themselves in sections 6.6.4, 6.6.5 and 6.6.6.

The tree manager also stores the syntax trees for the program, to enable it to perform the tasks for the user interface. The special macro functions (sMatch and sCreate) do not in fact need access to the tree, but the tree manager makes a good control object and localises all the information that is specific to the current transformation. It is both a server to the pattern functions, via sMatch and sCreate, and a client as it calls apply() for the top level pattern to start a conversion, with the parse tree as an argument.

The member functions for the user interface are readFile, to read in a new C/C++ file, doTrans, to perform a transformation, outCode to print a file to screen and outFile to save the current tree as code.

The user interface also needs to know the functions and structs that are available to become members, and tell the tree manager which of these the user has selected, before it starts the transformation. Finding and passing this information is covered in the next section.

### 6.6.2. Finding information for the user interface

To find the required information, a pre-pass of the syntax tree is performed before any conversions are done. A class derived from CpppTraverseActions is used to do this, and functions are defined that will be called whenever a FctDecl, StructDef or ClassDef node are found in the syntax tree. The template list class is used to store the nodes found, and so the member functions for this class are straightforward. The only check made is that the classes and structs are declared in the outer scope of the program, as otherwise no functions could take variables of that type as an argument.

C++ classes are found as well as structs because it may be desirable to add a new member function to a class, or to add member functions in stages when transforming a struct. This is stated in the requirements in section 3.7, but time constraints have prevented a pattern to find an existing class from being written, so ctrans does not support this type of conversion.

In fact, the user may wish to specify, as well as the class-to-be and the member-functions-to-be, the protection specification that the members of the struct should have in the new class. To allow this, the tree manager has a function that takes a struct from the list passed to the user interface, and returns a list of the member variables of that struct. This list will be passed back to the tree manager as an argument to doTrans(), with the protection spec of any variables required set. However, the user interface option to allow the setting protection specs has not been developed.

Storing a protect specification as well as a CpppAst node meant that the lists passed between the tree manager and user interface had to be of some container class, rather the just CpppAst nodes. It was also desired to isolate the interface from Cppp as much as possible, so new classes were defined for functions and structs as well as variables. These classes have member functions to return their name, so that the interface does not have to call CpppAst functions to print out a list of options for the user.

The object diagram for the MemVar, MemFn and StructClass classes resulting is shown in figure 6.7.

enum CtMemType  { func, variable }
enum CtVisibility { public, private, protected }
enum CtClassType { struct, class }

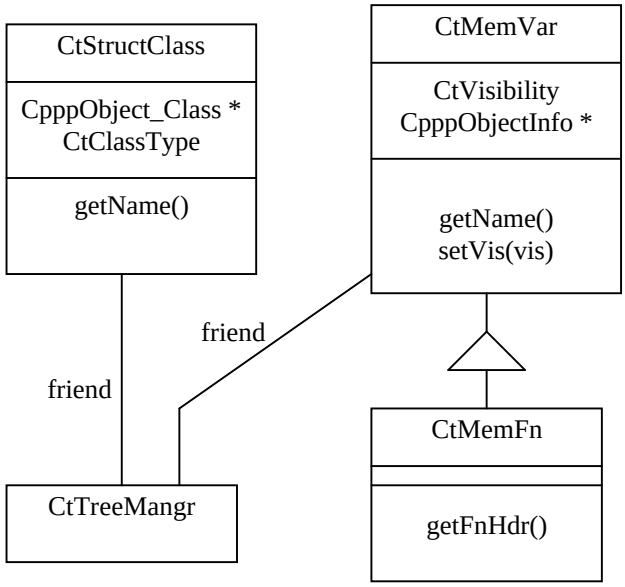



**Figure 6.7. Classes for storing global information**

### 6.6.3. Finding semantic information

The key semantic data required is firstly a unique identification for function names, used both to define and to call the function, and secondly the actual type of every identifier when it is used. This is as described in section 3.6.

This information can be made available by having a link from every SimpleName (identifier) in the parse tree to the CpppObject representing the variable or function in the semantic tree. Links are produced when the parse tree is duplicated, using a modified version of the existing copy() function for Cppp trees. This function is virtual, and is redefined for every Cppp class using a macro. The modification was to add a boolean argument, telling the function whether to add the links to duplicated nodes, and write a modified version of the function for three key classes.

Firstly, for a SimpleName representing the use of a variable, only the SimpleName copy() function needed to be changed. A new variable was added to the class, to store the semantic tree equivalent, and this equivalent node will have the required object as a son.

For function calls and definitions, the situation is more complicated, as the equivalent SimpleName node does not actually exist in the semantic tree. Instead it is replaced directly by a pointer to the object; however, this is not an insurmountable problem, as in both cases there is a node which is a common parent to the SimpleName. For function calls, the SimpleName in the parse tree and the object in the semantic tree are both sons of a CallExpr node. Therefore it is

possible to change the CallExpr copy() function, to add a pointer to the CallExpr in the semantic tree, but the pointer is added to its (newly created) SimpleName son. When access is needed to the object, the link can then be followed from the SimpleName, to the semantic tree CallExpr, to the object. The situation was similar for the function definition identifiers, using the link FctDecl->FctDeclHead->FctDeclarator->SimpleName in the parse tree, and FctDecl->Object_Function in the semantic tree.

To take advantage of the access to the unique objects, the tree manager also needs the objects for the CTB and MFTBs to compare them with. As the identity of the CTB and MFTBs are received from a subset of the list of structs and functions that are sent to the user interface, these lists should in fact be of the objects representing the structs and functions. To make this so, the pre-pass of the syntax tree is done over the semantic rather than parse tree, and the FctDecl, StructDef and ClassDef nodes found are all acceptable as they have links to the object.

Finally, extra information is required to recognise if a type specifier refers to the CTB or not. As found in section 3.6, a string comparison is sufficient and so the object is not needed, but typedefs for the CTB do need to be recognised. Cppp does not create links from a class to all the typedefs for that class, but rather links from the typedef to the class. Therefore, during the same pre-pass of the tree that gathers the functions and structs, links are created from every class or struct to typedefs for this.

To do this, a member function of the CpppTraverseActions class is defined that will be called for every Object_Typedef. If the typedef is for a class, or a pointer to a class, it is possible to follow links to the Type_Class node that represents the type of that class. This node is also accessible from the CpppObject representing the class.

Therefore a list variable has been added to the class Type_Class, and this is used to store CpppAst nodes representing typdefs.

### 6.6.4. Special matches

The special matches are checked by the sMatch() function, which uses the tree managers MFTB and CTB data. A list and brief description of the special matches is given below.

- match type names: DirCTB, PtrCTB, AnyCTB. These are used to check if the identifier passed is a type representing the CTB, a pointer to the CTB, or either. A string comparison is performed with the struct name, and all typedef-names for the struct. The input node may also be a ClassSpec, if the type is specified as “struct *struct-tag*”, and this is checked for.
- match CTB tag: CTBtag. If the tag exists for the CTB, this check is simple, but the tag for a struct may not exist. In this case, Cppp creates a PrivateName node with a unique number to represent it, but when this PrivateName is copied

for the duplicate parse tree, the copy() function creates a new PrivateName which will have a different number. Therefore PrivateName::copy() had to be re-defined to use a new constructor if the ‘do-links’ tag is set, so that struct tags will be the same in the parse and the semantic tree (from which the tree manager information is gathered).

- match the type of a variable: VarDirCTB, VarPtrCTB, VarAnyCTB. These use the object link added to each variable identifier to find if their type is the CTB. If the type of the argument SimpleName is &CTB, VarDirCTB will return true, and if it is &(*SimpleName), VarPtrCTB will return true.
- see if a variable is the CTB argument of a function: VarPtrCTBEq, VarDirCTBEq. Here, not only is the type of the variable checked to see that it is the CTB, but the variables name is also checked to see if it is the same as that of a second argument.
- member functions: MFTB. The same macro can be used for the calling and the definition of a functions, as the object is son[0] of the semantic node stored for both cases.
- find if a class variable is to be public or private: MemPri, MemPub, MemProt. This checks the variable name against the name of all variables in the list of CTB members. If the correct variable is found, the desired protect specification for that variable is looked up in the MemVar class variable.
- displaying error messages: CallError, AssignError. Used just to display a message if an assignment is found to the CTB argument or a member, in a function where the CTB argument is by value.

### 6.6.5. Special create - CTBName

The name of the new class is used consistently in the new program by using this special create to return a SimpleName. In fact, sCreate just returns a ‘specified_name’ variable of the StructClass that holds the CTB. This variable was declared so that the user could set the class name to be any name of their choice, but the again the user interface does not have an option for the user to set this. However, ctrans generally chooses the most appropriate name, which is set when each class is converted into a StructClass variable in TreeMangr::readFile. If there is a typedef for the CTB (but not for a pointer to the CTB) this becomes the name, and otherwise the struct tag is used.

### 6.6.6. Special create - MFTB Headers

The only other special create is MFTBHdrs, used to create the prototypes for the member functions to be placed in the class definition. It was thought that it may be difficult to produce the MFTB prototypes with the pattern matching approach, but using a special macro is an elegant solution, and allows access to the list of MFTBs held by the tree manager.

The problem found was that it is the object that is stored for these MFTBs, and not the parse tree node that is used to convert the definitions of the MFTBs. As was the case for the whole function (see section 6.4.3), it proved impossible to convert the function header from the information available in the object, especially as the prototype has to agree with the function definition.

To resolve this, it was chosen to add a duplicate of the FctDeclHead from the parse tree to the Object_Function in the semantic tree. Using the parse tree node in the MFTB lists had been considered, but then the program would not have had any access to the CpppObject for the function. Also changing the copy() function had investigated, but this would require a link to be made from the original node to the duplicate. As the copy() functions are declared as 'const', and advantage of this is taken by the rest of ctrans to duplicate the constant pattern nodes, this option was rejected.

The difficulty with the chosen option was that the semantic analysis functions in Cppp had to be modified. However, in this case the algorithm did not have to be understood, as the preAction and postAction semantic functions could both be modified to achieve the desired result. A duplicate of the FctDeclHead node was made in FctDecl preAction, before it was replaced by an object, and this duplicate was added as a new son to the object after all the semantic actions had been performed.

The node was added as a son, rather than just creating a new pointer variable in the Object_Function class, to ensure proper de-allocation of memory. This is because the CpppAst destructor only de-allocates the memory for its sons and itself, and not for any other pointer attributes it has. Previously, when new variables had been added to Cppp classes, there were links to them from elsewhere so they would be deleted, but this was not the case here as the tree section added was a duplicate. This is important as it is the only change that has been made to Cppp that alters the tree that it produces, but only the semantic tree is different, so ctrans is not affected by this.

This means that the syntax tree section used to produce the headers in function definitions is available to the sCreate function, and so it made sense to use the same conversion algorithm. Therefore a special set of patterns to apply to the FctDeclHead of every MFTB was created, and these are merely the section of the main MFTB patterns that apply to the function header. The MFTBHdrs special create iterates through all the MFTBs, calling apply() on the 'mftb_hdrs' pattern, and adding the result as a son of the result from the special macro.

### 6.6.7. Summary of tree and Cppp use

An overview of the tree managers tasks is shown in figure 6.8. This shows the action of the functions readFile() and doTrans(), when an input file is read in and a transformation performed on it.

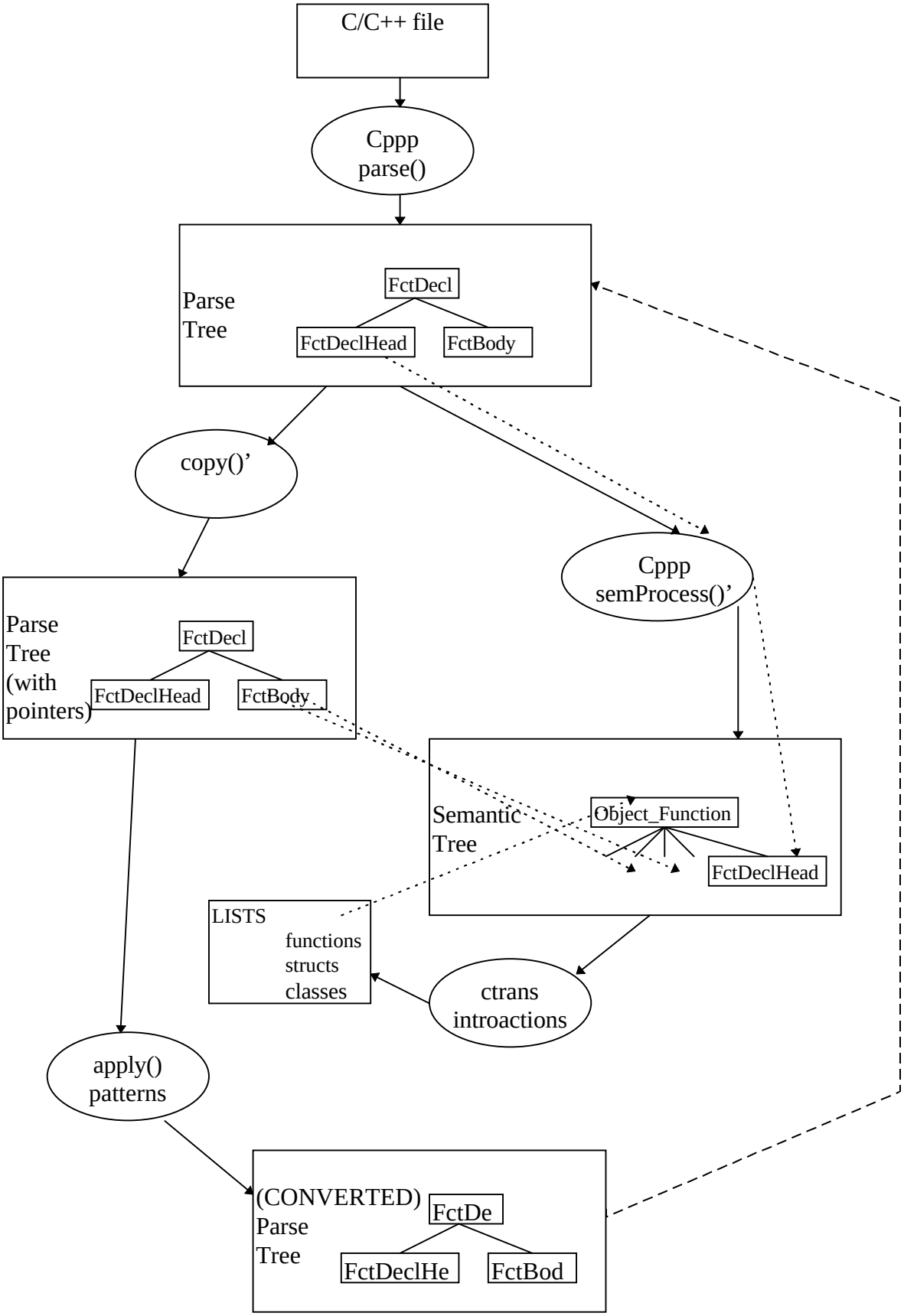



**Figure 6.8. Trees and Cppp use**

Starting with readFile(), this calls the pre-processor, and then calls the Cppp parse() function to create the syntax tree for the input program. This tree is then copied, using the modified Cppp copy() routine, which adds links to certain nodes to the original tree, as described in section 6.6.3. Next the Cppp semantic processing function is called on the original tree, to create the semantic tree. This tree is no longer in a form that allows it to be transformed, but it holds certain semantic information that is required. Also, the semProcess() function in the diagram is shown with a dash, because it has been modified to add a copy of the each functions original FctDeclHead the Object_Function.

The final step performed by readFile() is a traversal of the semantic tree to find and store all the structs and functions in the program. The semantic tree is used for this so that semantic information is available for whichever of these become the CTB and MFTBs. This is also why the copies of the original FctDeclHead are needed - these objects are the only data available when the prototypes for each MFTB have to be produced in the new class definition. As the prototype cannot be created from the CpppObject, a copy of the parse tree version of the header is used instead.

The tree manager is storing a parse tree (with pointers to the semantic tree), a semantic tree and the lists at this stage. doTrans() must be called next, and this function applies the top-level pattern to the parse tree. The transformed parse tree results, and then the information gathering is repeated to prepare for a possible further transformation. Therefore, a duplicate of this tree is made that will become the new parse tree, and then semantic processing is done on the modified tree, so it becomes the new semantic tree. Finally the Introactions traversal of the semantic tree is performed again, all from the doTrans() function.

## *6.7. Re-scanning*

### 6.7.1. Checking against other nodes

This section is about the need that sometimes arises to check over a section of syntax tree twice. The implementation of the feature to allow this is covered in section 6.7.2, and the implications that this has for the numbering of star nodes are described in section 6.7.3.

A re-scan of the syntax tree is sometimes necessary when the SimpleName node being examined has to be tested against another node from the tree. A special match, EqualTo(X), is used with two node arguments to find if they are the same. One of these arguments is the syntax tree node equivalent to the if-find-node currently being tested, and the other is specified by a star number. The node corresponding to this star number will (usually) have been found in an earlier part of the syntax tree, but if it has not been encountered yet, a second scan will be required.

Checking if one node is the same as another is necessary in MFTBs that return a CTB variable, as described in section 3.3.2. This is because the declaration for the local variable that is returned has to be removed. This variable is identified by finding the return statement, but this statement will occur after the declaration.

The other case when nodes have to be compared against each other is in the function body of all MFTBs, to find out if a SimpleName is the CTB-argument variable. As described in section 3.3.2, as well as checking if the identifier is the same, the type of the first node also has to be checked to make sure it is the CTB. The special macros VarPtrCTBEq(X) and VarDirCTBEq(X) are used for this, but in this case the CTB-argument will have already been declared, and so no re-scan should be needed.

### 6.7.2. ReScan

This section describes how this second scan of the syntax tree is implemented, and how the need for such a scan is detected.

It will only ever be necessary to check over a section of tree again if a special match is used with an argument. As the sMatch() function is called from IfFindNode::isSame(), isSame will determine if a re-scan is needed. This function will look up the argument star-number in the star-list, and if no node with this number is found, the star-list will return NULL. A re-scan will then be required.

The key implementation question is, how much of the syntax tree needs to be searched to find the required node? This question will be examined with reference to figure 6.9. In figure 6.9 b), part of a syntax tree for a function returning a CTB variable is shown, and in a), the pattern for removing the declaration of this variable is shown.

In the pattern, the if-replace-specs irs#1 and irs#2 would both be attached to an if-find node representing the FctBody (along with, for example, if-replace-specs for converting data member accesses). When the ObjDecl is encountered, and a re-scan tag is set, the remaining nodes in the FctBody must also be scanned to find the desired star node. This first pass now serves only as a stage to find the missing star number, and the second pass will perform any required modifications to the tree.

Therefore, rather than returning a boolean, isSame() returns Yes, No or ReScan. If an activation of isSame receives a result of ReScan from a recursive call, it returns ReScan to its caller; but first, it continues checking its sons. This is in case the required node is actually in the same if-find pattern, as shown in figure 6.10 a) . An if-find patter such as this does not occur in any of the patterns written so for, but it could occur in a future pattern, and so it should be allowed for.

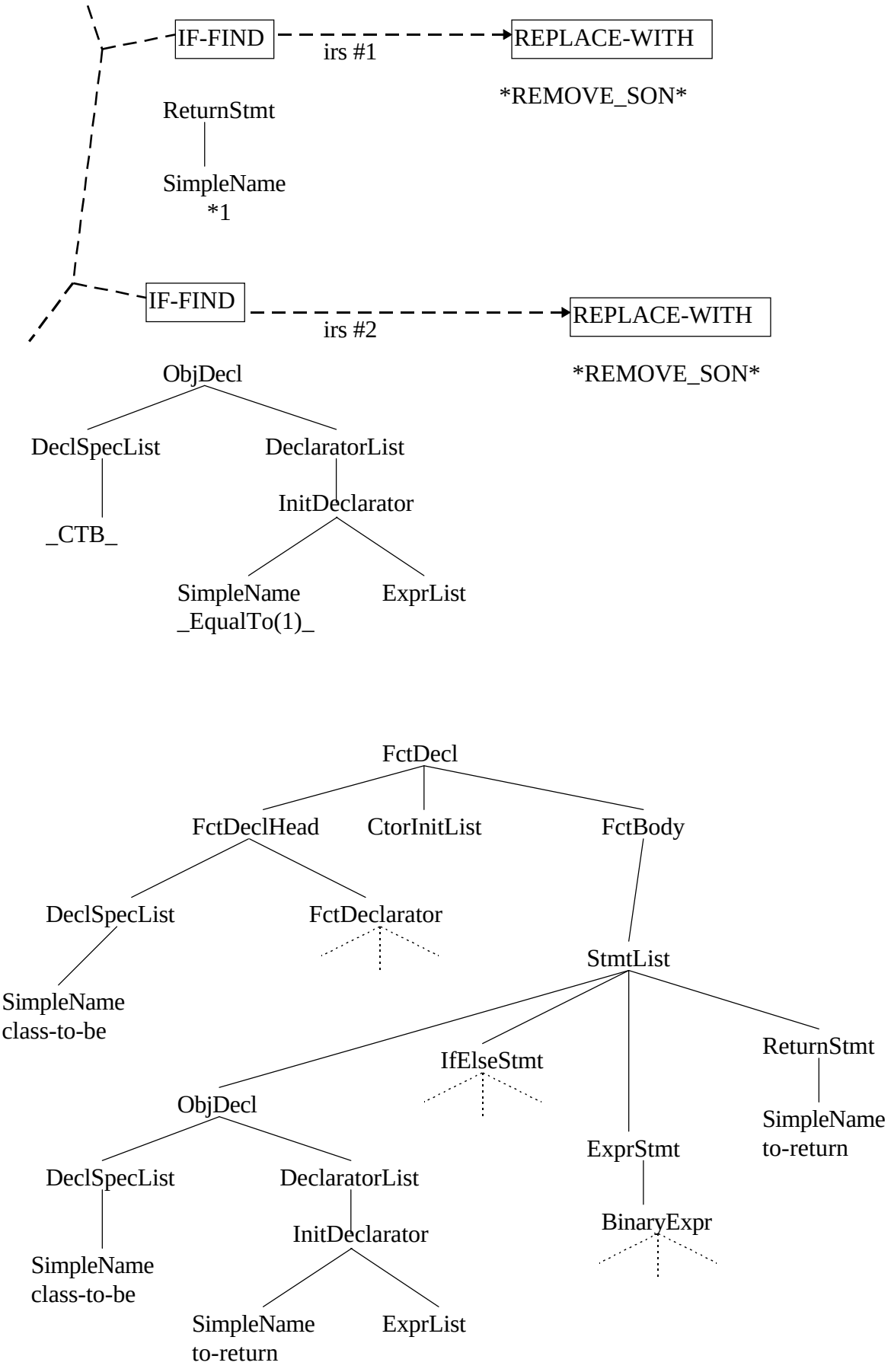
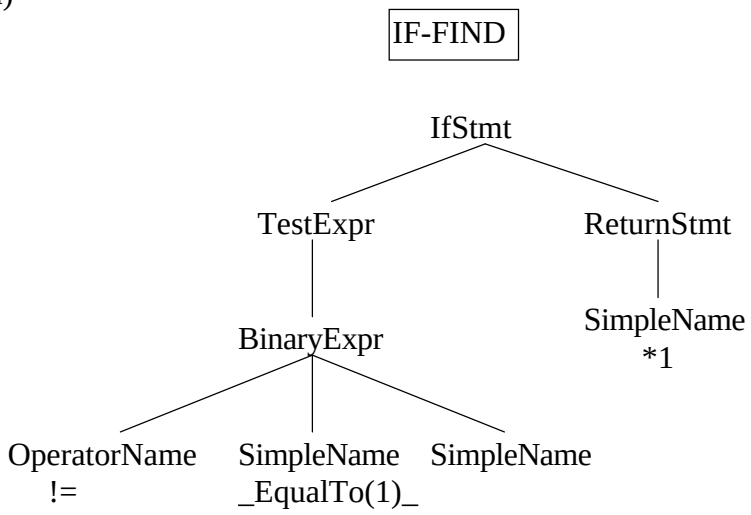
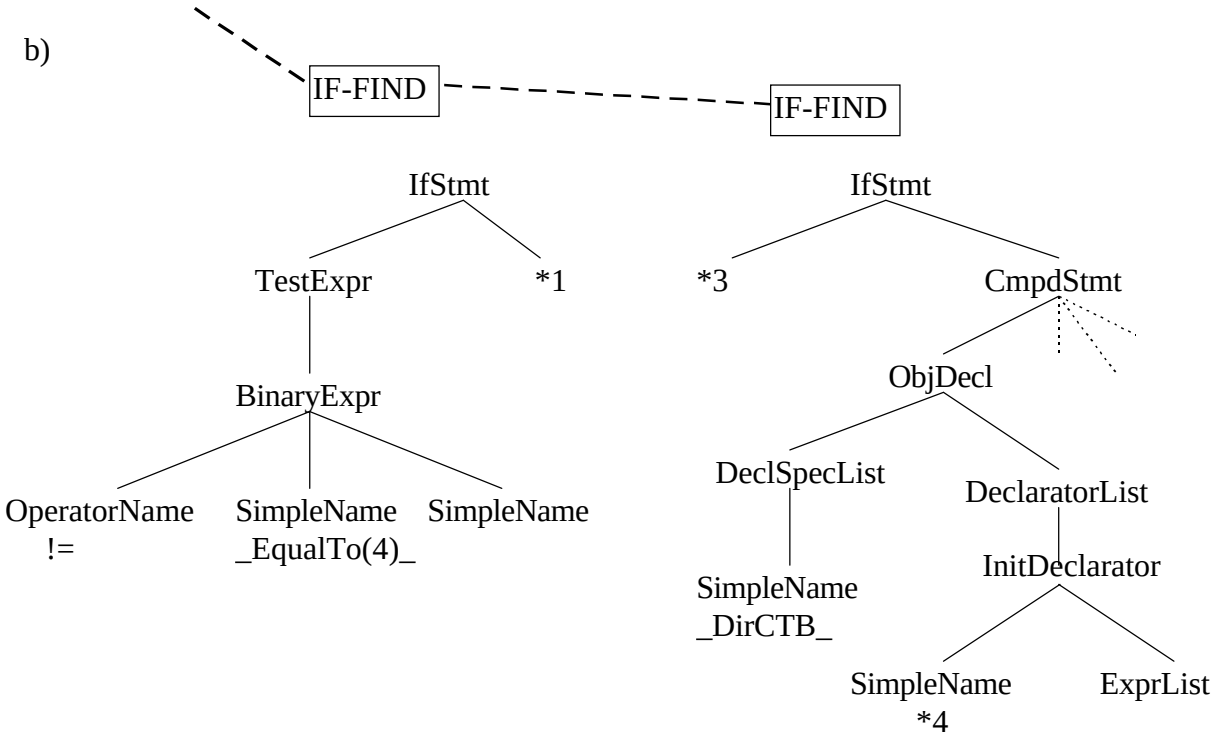



**Figure 6.9. Re-scan example**

a)


IF-FIND
IfStmt
TestExpr
ReturnStmt
BinaryExpr
SimpleName
*1
OperatorName
!=
SimpleName
_EqualTo(1)_
SimpleName

b)


IF-FIND
IF-FIND
IfStmt
TestExpr
*1
BinaryExpr
OperatorName
!=
SimpleName
_EqualTo(4)_
SimpleName
IfStmt
*3
CmpdStmt
ObjDecl
DeclSpecList
DeclaratorList
SimpleName
_DirCTB_
InitDeclarator
SimpleName
*4
ExprList

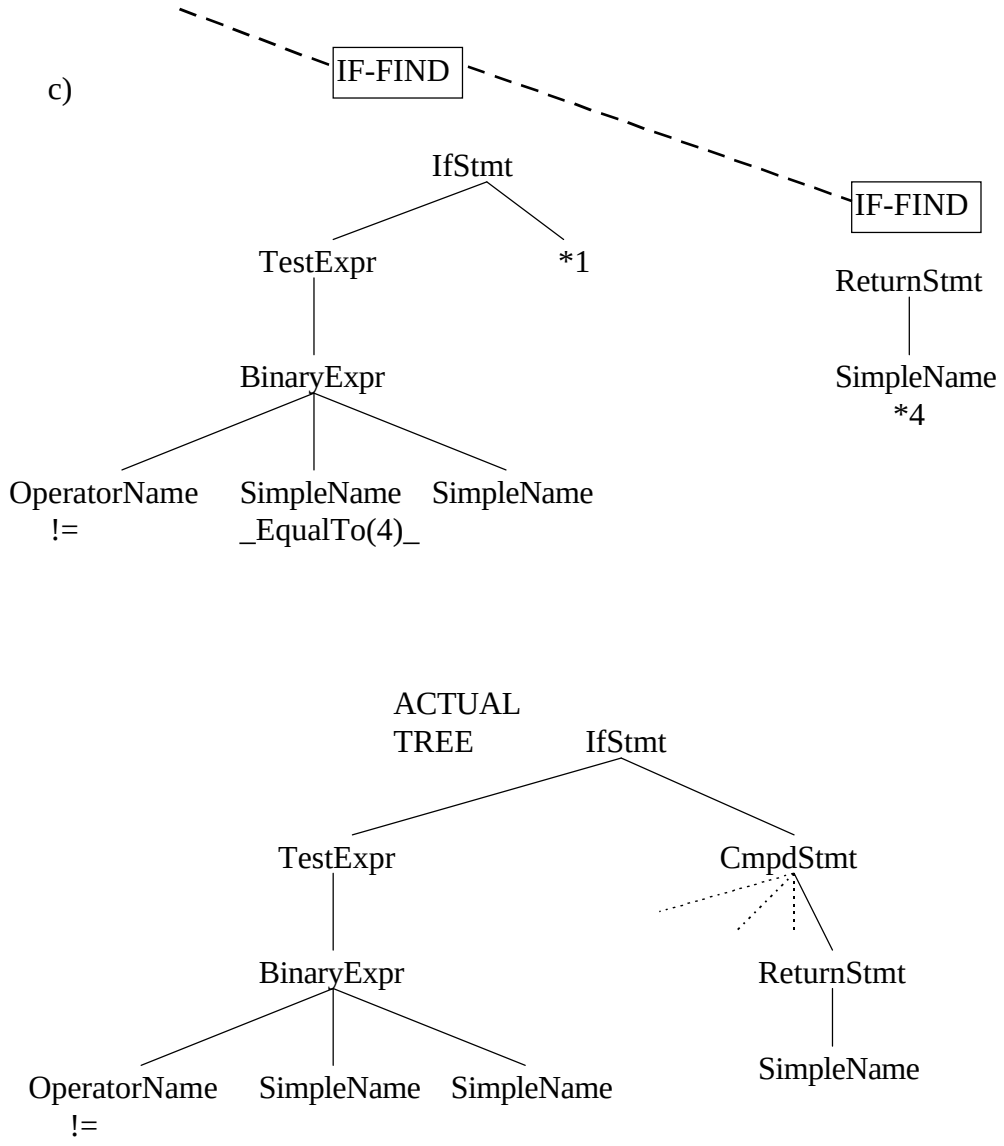



**Figure 6.10 Possibilities with re-scan**

Eventually, the ReScan result will be returned to an activation of TransSpec::doApplication, and in figure 6.9, this activation will have been called for the ObjDecl node. It can be seen that the action of this function must not be simply to re-apply all the if-replace-specs to the node it was called for, as it would not find the return statement as a son of the ObjDecl node. Therefore doApplication must also return ReScan, but before this it should continue with various checks. Firstly, it must continue to check the other if-replace-specs on the root node, to allow for cases such as the one shown in figure 6.10 b) (again hypothetical). Secondly, it must test itself against the sons of the root node, in case a situation like the one shown in figure 6.10 c) is found.

Finally, referring to the example in figure 6.9, the doApplication activation for StmtList will receive a ReScan result. This activation must continue

to call itself recursively for its other sons, to find the ReturnStmt, and then return ReScan itself.

The ReScan result will then be received by either the apply() or matches() functions, and they can then call doApplication again. If this call also returns ReScan, an error message is printed.

### 6.7.3. Extent of star-number visibility

This section deals with the scope of a star number within a pattern. It would be logical for the scope to be restricted to the if-replace-spec in which the star-number is 'declared', which is the one containing the if-find node that will cause it to be stored in the star-list. This would enable pattern writers could start numbering the stars from '1' in every "if-find -> replace-with", and not worry about potential clashes. Unfortunately this is not possible, because the identifier for the CTB-argument of an MFTB needs to be tested against nodes found in the MFTB function body. Therefore star numbers need to be accessible from any trans-spec that is a child of the if-replace-spec in which they are declared.

This means that the star-list cannot be local to a TransSpec member function. However, there are problems with having the star-list global in ctrans as well. This would make it tricky to write patterns, as every number in every pattern must be different, but also it would cause errors if more than one occurrence of a particular IfReplaceSpec was found. Then different nodes will be stored with the same number, so the wrong node may well be received from the star-list and used in the transformed syntax tree. Multiple occurrences of an if-find pattern are quite common, for example there will often be more than one access to a data member within an MFTB.

This suggests that star numbers should be visible in the if-replace-spec that they are declared in, and throughout any child trans-specs that it has. However, this is further complicated by the need to find the declaration of a returned variable, as discussed above. In this case the nodes to be compared are declared in if-replace-specs with the same trans-spec as a parent. The resulting requirement, which is to have a star number visible throughout the trans-spec in which it is declared, contradicts the requirements above regarding multiple matches of an if-find pattern.

The final solution is to make it the responsibility of the pattern writer to decide when star numbers are no longer needed. The pattern writer must specify a list of star numbers to remove, which will be stored with each IfReplaceSpec. These will be removed from the (completely global) star-list after the application of isSame(), or after the application of isSame(), freeAndModify(), and create() if the isSame() call is successful. The freeStars() member function for IfReplaceSpec is responsible for removing them, and this is called from TransSpec::doApplication. Note that it only removes the CpppAst nodes from the list, it does not de-allocate memory for them.

Usually, the pattern writer will remove all the stars that the if-replace pattern uses, in which case they will be visible only in that pattern, and in any child trans-specs. However, if access to a particular star node will be required from if-replace-specs belonging to the same trans-spec, the number of that node can be left off the remove list. In this case, it will be visible throughout the trans-spec it is declared in, and possibly in part of the parent if-replace-spec. Also, the number should be placed on the remove list of the parent if-replace-spec, so that it is eventually removed.

### *6.8. Remove-son and Concat*

These extensions will be considered together, as they both require modification to be made to a parent node that is unavailable to the pattern that requests the modifications. The abbreviation C/RS will be used to mean CONCAT / REMOVE_SON.

Firstly, the method used for marking a node as C/RS will be discussed. Next, the point at which modifications will be made to a parent node will be explained, and finally, some details of the modification algorithms will be given.

As C/RS nodes will be returned from ReplaceNode::create(), they must be of type CpppAstInfo (or a derived class). One way of marking such nodes would be to use SimpleName nodes, and set the identifier name field to *REMOVE_SON* or *CONCAT*. However, this would have required multiple checks every time a SimpleName node was found, and so an alternative approach was used.

This was to use CpppAst nodes whose type was *REMOVE_SON* or *CONCAT*, but creating such a node was not straightforward, as the type name variable is set in the constructor of each node class, declared 'const', and a private variable of the base class. Therefore a new class (CpppAst_CtransDummy) was defined, for which it was possible to set the node id to be any string.

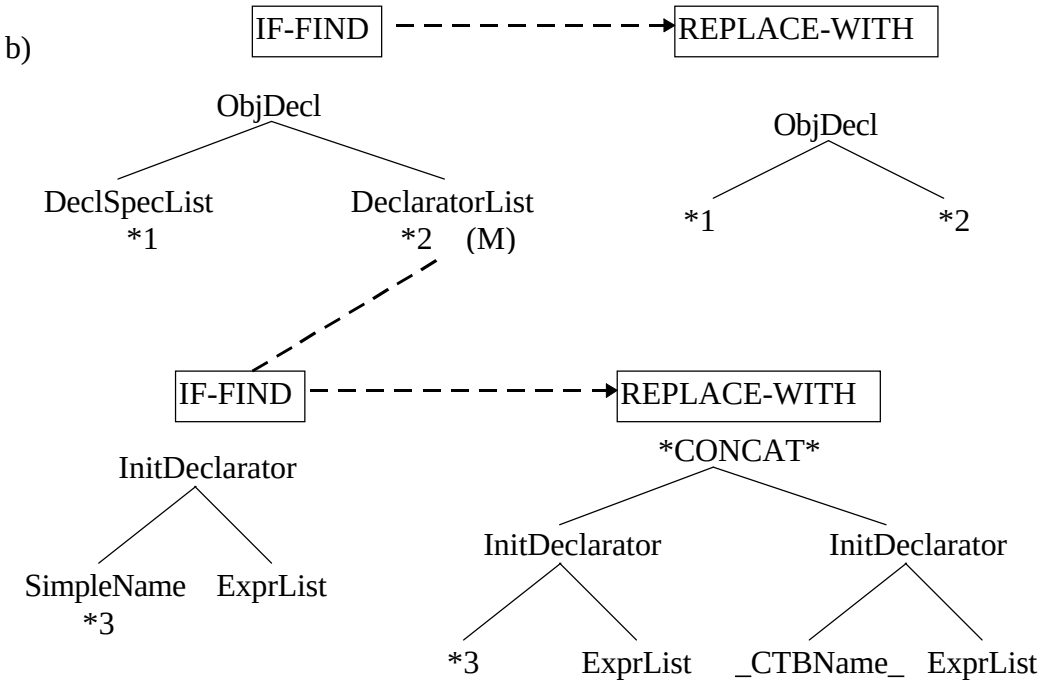
The essential problem leading to the need for C/RS nodes is that for the if-replace-specs that they are used in, the parent of the input syntax tree node needs to be modified. This means that the modifications cannot be made in ReplaceNode::create() when these nodes are created. The obvious solution is to look at the procedure that calls this, which is TransSpec::doApplication. When this is called recursively on the sons of its input node, the results can be checked to see if they are C/RS. At this point, the required parent node is available to be modified.

However, this will not work if it is the first call of doApplication that returns a C/RS node, and not a recursive call. In this case, the apply() or matches() functions that will call doApplication will not be of any use, as they are called with the same root syntax tree node as they pass to doApplication.

a)

```
struct s_tag {int I }
const char  conv, other =  'a' ;
```

b)




c)

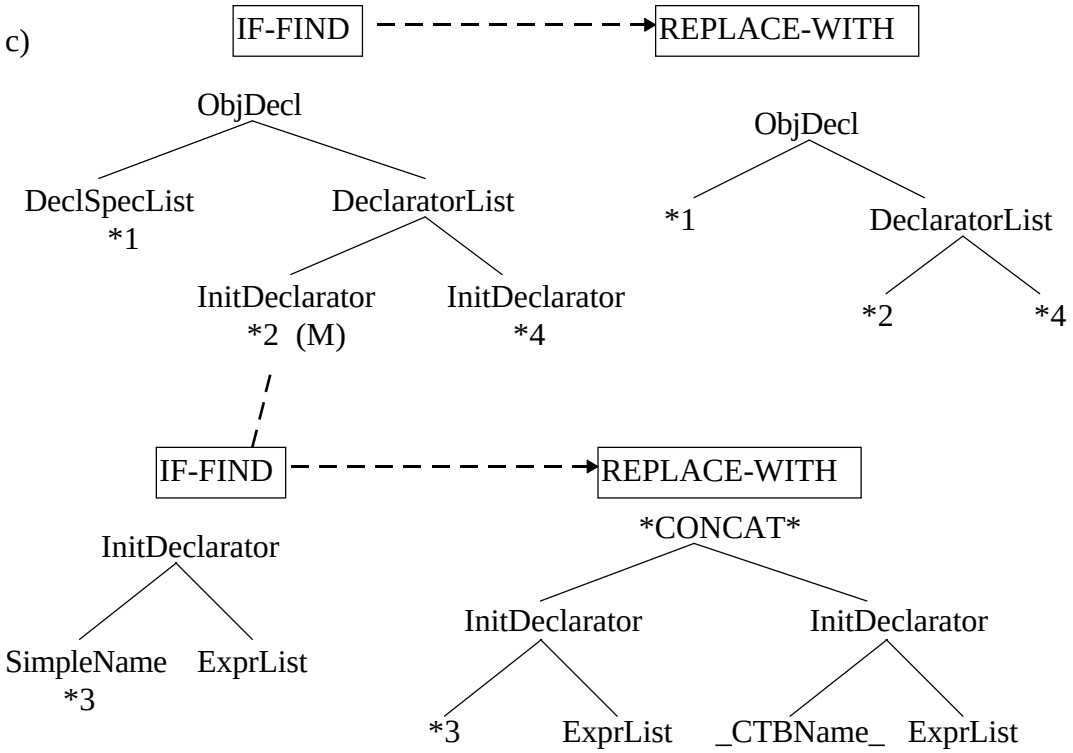



**Figure 6.11. Testing patterns for CONCAT**

The node returned by apply() will either be the root node of the output tree, or will at some point be incorporated into a larger tree within the create() procedure. Therefore, create can be used to 'expand' the C/RS nodes, by checking the result it receives when it calls itself recursively for its sons. An example of a case when this would happen is shown in figure 6.11 c), and it can be seen that the node will in fact reach create() via the star-list.

Figure 6.11 shows a short program and two patterns that are in fact part of the test set used for testing C/RS features. In the pattern shown in b), the C/RS will be expanded by doApplication(), and in the pattern shown in c) it will be expanded by create(). Analysis has shown that checks in both procedures are necessary, and that instances of C/RS will always be expanded, except if they are the root 'Program' node of the output tree (which would not make any sense).

Now the algorithms for expanding C/RS will be covered briefly. Within the create() procedure, expansion is easy, as the returned node has no sons to start with. If the result of create() called for the ReplaceNode's sons is REMOVE_SON, then the result node is not added to the new node, and if the result is CONCAT, then the sons of the result node are added to the new node.

Within doApplication, the sons of the input node are modified, rather than created from scratch. If the result from doApplication called on (existing) son number X is REMOVE_SON, then son X must be removed from the sons array of the syntax tree node. If the result is CONCAT, then the sons of the result need to be added to the tree node, starting at position X in the array. This means that the existing sons need to be copied to the next location in the array every time a son is inserted, and a routine to do this had to be written. Finally for the CONCAT algorithm, the original son X has to be removed from the array.

## *6.9. Interface*

Two main components make up the user interface in ctrans, a command interpreter and a CtInterface object, which will be covered first. The interface object manages communication between the tree manager and the command interpreter, as shown in figure 6.12.

The main task of the interface object is to set up a pointer to the CTB, and a list of the MFTBs, to be converted. The member variables of the object are six key data items, which are lists of all the functions, structs and classes in the input program, a list of all the MFTBs, the CTB and a list of all the member variables of the CTB.

The first three lists are set up by the tree manager whenever it reads a new file, and are in fact lists of CtMemFn and CtStructClass objects. This is so that the interface does not have to access CpppAst nodes, but can use member functions of these classes for displaying the names of these classes and functions.

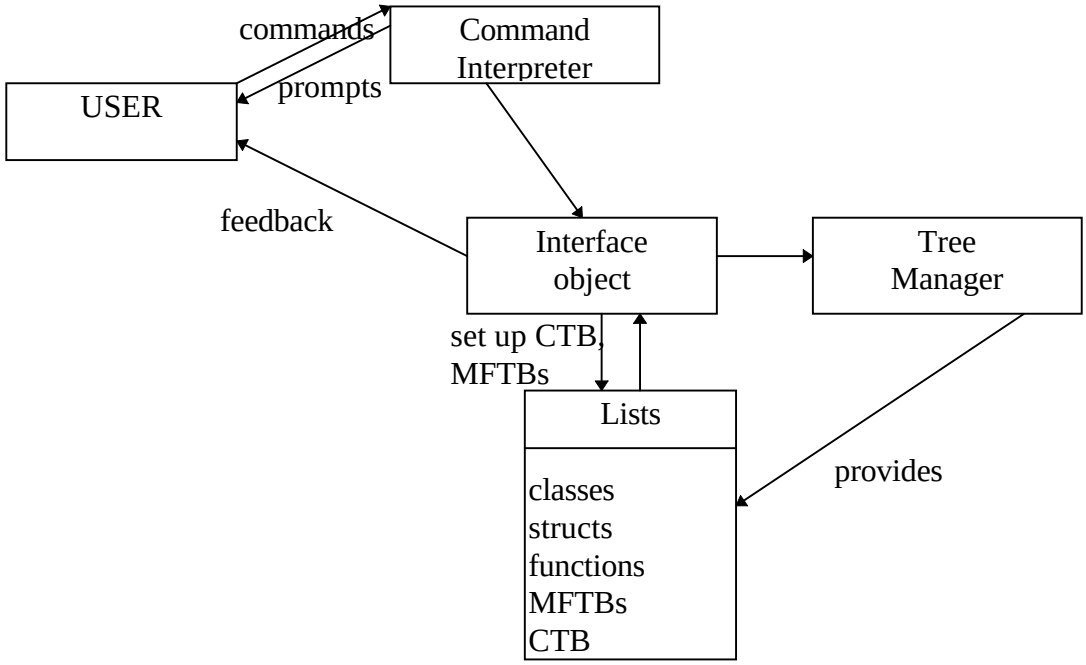



**Figure 6.12. User interface management**

The MFTB list and CTB are set by the user, and are CtMemFn and CtStructClass objects taken from the lists produced by the tree manager. The list of member variables is kept so that the user can set the protect specification of these (although the command interpreter does not support this option). When the user sets the CTB, the interface passes this list to the tree manager, to add the variables belonging to the struct chosen. Pointers to these three data members are passed to the tree manager when the doTrans() function is called.

For the use of the command interpreter, the interface object provides functions for setting and adding to these three data structures, and also functions for displaying the original three lists of all the functions/classes in the program.

The interface object is in fact a local variable of the command interpreter module. This module is straightforward, and is fundamentally just a loop that calls the interface object to execute commands, until the user quits. For some commands, the system will enter a mode to request further information from the user, and these are managed by the command interpreter. An example is prompting the user for the name of the CTB, and then of MFTBs, when the ‘convert’ command is entered.

There are also several ancillary functions, for example to read and break up into words a line from the standard input, and a function to check if an input string is the same as, or a subset of, an input command string.

### *6.10. Pattern translator*

The patterns shown in this report have been represented graphically, as trees, but C++ objects must be declared to represent these patterns for them to be used by ctrans. These objects were created for one pattern manually, and it was found that this was a very lengthy and tedious task. Generally, two declarations are required for each object as the array to store the sons has to be declared separately, and the names of each node's sons have to be kept track of. Also, most of the arguments to the constructor were NULL for any one node, and it was tricky to ensure that the particular components that a node does have, were being passed as the correct argument.

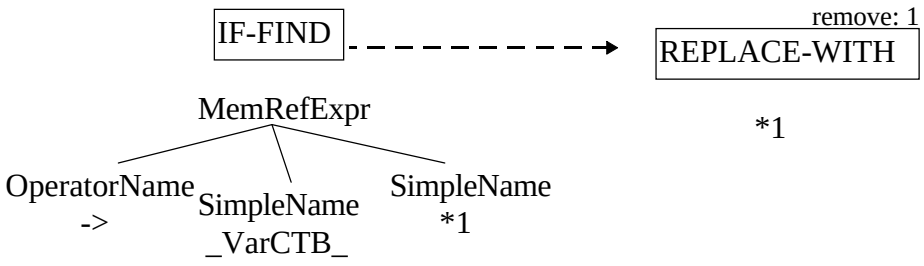
It was decided that a tool to automate this conversion from abstract pattern to C++ 'const' objects was needed, and so pt (for 'pattern translator') was designed and implemented.

The input for pt is a file written using a language inspired by Cppps textual representation of syntax trees. The ctp (CTrans-Pattern) language requires each node to be declared on a new line, and the sons to be declared on lines below it, indented by one level more. This means that indentation must be used absolutely consistently in the input file, or else it will be interpreted incorrectly.

An if-find or replace-with node is represented by just listing its components, with special macros bracketed by underscore characters (_MFTB_), star numbers prefixed by an asterisk, and the trans-specs indicated by a C and/or M tag enclosed in parentheses.

Trans-specs are treated as a son of the if-find node that owns them, and are defined by a line containing "#TransSpec". Similarly, if-replace-specs are treated as sons of a trans-spec, with the identifier "#IfReplaceSpec". These also have the number-to-find, and then a list of any star-numbers to remove, following the # identifier. An example of a pattern is given in figure 6.13, where a) is the graphical form and b) is the ctp form.

a)



b)

```
#IfReplaceSpec  -1    1     //   '-1' means no restriction on num-to-find
   MemRefExpr
      OperatorName  ->
      SimpleName   _VarCTB_
      SimpleName   *1
   *1
```

**Figure 6.13 Example of ctp syntax**

The number of sons a node or trans-spec has does not need to be specified, although the constructor will need to know, and no markers to distinguish if-find nodes from replace-with nodes are needed either. Comments are allowed in C++ format, and can be placed anywhere.

Each object or initialised array printed by the program has a unique number, and its name is this number preceded by the standard prefix for its type, as described in section 6.5.2. The starting number for object names is supplied by the user, on the command line.

A recursive procedure will be used to output the sons of each node, and this will lead to the sons being printed before the parent node. Although this makes the output code hard to understand, the C++ compiler needs to have the sons declared before they are referenced in the sons-array. The ctp files are intended to be used for maintenance, rather than the code files, so this is not a problem at all.

Producing the objects is handled by two main functions, doObject() and readLine(). doObject is the key, recursive, procedure. It relies on two global variables: the node number, and the next indent, which it will update before returning or making a recursive call. The function returns an integer which is the id-number of the object it prints out.

The first thing that doObject() does is call the other main procedure, readLine(). This reads in one line from the input file, and splits this up into separate words. It then identifies the node component that each word represents, and stores them in appropriate variables. It also returns whether the object read in is a node, trans-spec or an if-replace-spec.

doObject() then reads in the indentation level of the next line, and stores the original indentation. If the new indentation is less, then this node has no sons, and no recursive call is made. In this case, a function to output the node is called.

Otherwise, recursive calls are made to read in the sons of this node, and the number of each of these (returned from doObject()) is stored, so that it can be output in a sons-array.

For if-replace-specs, there will be two sons: as these must be an if-find and a replace-with node, the recursive calls are made with tags set to indicate this. This is how the program knows whether it should print an IfFindNode or a ReplaceNode.

One of the checks performed on the input is whether a tab character is found; if it is, the indentation level will appear different to the pattern writer, and the pattern will be represented incorrectly. Therefore an error message giving the line number is printed and the program aborts. Other checks made are that only valid special macros are used, and that a replace-with node exists for every if-find node.

The two functions doObject() and readLine() are defined in the file ctmain.cc. The only other files used in pt are ptoutput.cc and the header file for this, and these contain a functions for printing each of the object types, and each of the sons-array types. These functions are not very intelligent, and basically just print the name of the constructor function and then all the arguments for that constructor.

The ReplaceNodes are worth mentioning, however. These must contain a pointer to a CpppAst node, and these nodes should ideally be constant as well, rather than wasting memory by creating a new node each time. Therefore a const object is declared for every CpppAst node required in a replace-with pattern, in the file ctastpatterns.cc. These are declared in the header file ct_patterns.h, and this file is automatically ‘#include’d into the C++ file produced by pt. All of the const CpppAst nodes are named “Cp” followed by the node type name (e.g. CpSimpleName), and these names are produced by the output function.

Finally, pt will save the C++ file produced as “*filename*.cc”, where “*filename*.ctp” is the name of the input ctp file.

### *6.11. Integration and testing*

Unfortunately, due to time constraints, the testing phase for ctrans was very brief.

Initial testing was performed for a basic ctrans shell, without any of the pattern matching engine in place. The modules that could be tested were, a early version of the user interface, the generic list class, the CpppTraverseActions (intro-actions) for creating lists of the functions, structs and classes in an input program, and the calls to the Cppp parsing function. Tests were performed to check that this shell was able to create a syntax tree for an input program, and find and print a list of the functions found in it.

Further testing had to wait until all of the pattern matching engine had been implemented, and these tests were more thorough white box tests. A small program, of 5 or 15 lines, was written to test each of 8 key aspects of the application of patterns, and these aspects are listed below.

- Re-scan
- CONCAT
- REMOVE_SON
- visibility of star-numbers
- special matches (with / without argument)
- special create
- num-to-find for if-replace-specs
- finding more than one if-replace-spec

For each, a series of patterns was written to test all the sections of code used to implement feature. An example of two of the patterns, and the short program, used to test the CONCAT feature is shown in figure 6.11. (Essentially the same patterns were used for REMOVE_SON testing also). With these tests, several bugs were found in the pattern matching engine, but probably more were found in the patterns, even for such small examples.

Further testing was performed on the complete C example programs that were used for analysis. Firstly transformations were tested with just two complete patterns, and then a complete set of ten patterns was added and tried on the examples. These tests revealed several minor problems, some of which were fixed, and for almost all of the others, the solution is known but has not been implemented due to time pressures.

## *6.12. Results*

### 6.12.1 Array-based queue

- Input program

```
#define MaxQueueSize 50


typedef struct queueCDT * queueADT;

struct queueCDT {
      void *array[MaxQueueSize];
      int len;
};

queueADT NewQueue(void) {
   queueADT queue;
//   queue = New(queueADT);   cppp parses this wrong
   queue = (queueADT) malloc(sizeof(queueADT));
   queue->len = 0;
   return (queue);
```

```
}


void FreeQueue(queueADT queue) {
   FreeBlock(queue);
}

void Enqueue(queueADT queue, void *obj) {
   if (queue->len == MaxQueueSize) {
      Error("queue full");
   }
   queue->array[queue->len++] = obj;
}

void * Dequeue( queueADT queue) {
   void * result;
   int i;

   if (queue->len == 0)  Error("removing from an empty queue");
   result = queue->array[0];
   for (i=1; i < queue->len; i++)
      queue->array[i-1] = queue->array[i];
   queue->len--;
   return (result);
}

int Queuelength(queueADT queue) {
   return (queue->len);
}
```

- User input

  CTB: queueCDT
  MFTBs: NewQueue, FreeQueue, Enqueue, Dequeue, Queuelength

- Converted program

```
typedef  struct queueCDT *queueADT;

class queueCDT {
   private:
      void  *array[50];
      int  len;
   public:
      queueCDT();
      void  FreeQueue();
      void  Enqueue(void  *obj);
      void  *Dequeue();
      int  Queuelength();
} ;

queueCDT::queueCDT() {
   len = 0;
}

void  queueCDT::FreeQueue() {
   FreeBlock(queue);
}

void  queueCDT::Enqueue(void  *obj) {
   if (len == 50)  {
      Error("queue full");
```

```
   }
   array[len++] = obj;
}

void  *queueCDT::Dequeue() {
   void  *result;
   int  i;
   if (len == 0)
      Error("removing from an empty queue");
   result = array[0];
   for (i = 1;
        i < len;
        i++;
        )
      array[i - 1] = array[i];
   len--;
   return result;
}

int  queueCDT::Queuelength() {
   return len;
}
```

- Remarks

This program is from Roberts [10].

There are only two errors in the converted program:

In FreeQueue(), the call to FreeBlock has not been changed to use ‘this’. This is due to a bug in Cppp, which is that there is no link to the object from this SimpleName.

In Dequeue, the for loop has an extra semicolon after ‘i++’

### 6.12.2. Dynamic stack

- Input program

```
// #include <stdio.h>
#define NULL 0
typedef int Item_type;

typedef struct node_tag {
    Item_type info;
    struct node_tag *next;
}  Node_type;

typedef struct stack_tag {
    Node_type * top;
}  Stack_type;


void Error(char *s) {
  printf( "%s,\n", s);
};
```

```
Node_type  *MakeNode( Item_type item) {
  Node_type *p;
  if ((p = (Node_type *) malloc(sizeof(Node_type))) == NULL)
     Error("out of memory");
  else {
     p->info = item;
     p->next = NULL;
  }
  return p;
};


void PushNode( Node_type *node_ptr, Stack_type *stack_ptr) {
  if (node_ptr == NULL)
    Error("Attempt to push non-existant node");
  else {
    node_ptr -> next = stack_ptr -> top;
    stack_ptr -> top = node_ptr;
  }
};

void PopNode(Node_type **node_ptr, Stack_type *stack_ptr) {
  if (stack_ptr->top == NULL)
    Error("Trying to pop from empty stack");
  else {
    *node_ptr = stack_ptr->top;
    stack_ptr->top = (*node_ptr) -> next;
  }
};


void Push(Item_type item, Stack_type *stack_ptr) {
   PushNode(MakeNode(item), stack_ptr);
};

void Pop(Item_type *item, Stack_type *stack_ptr) {
  Node_type *node_ptr;
  PopNode(&node_ptr, stack_ptr);
  *item = node_ptr -> info;
  free( node_ptr );
};
```

- User input

  conversion I:
  CTB: node_tag
  MFTBs: MakeNode

  conversion II:
  CTB: stack_tag
  MFTBs: PushNode, PopNode, Push, Pop

- Converted program

```
typedef  int  Item_type;

class Node_type {
   private:
      Item_type info;
```

```
      struct node_tag *next;
   public:
      Node_type(Item_type item);
} ;

class Stack_type {
   private:
      Node_type *top;
   public:
      void  PushNode(Node_type *node_ptr);
      void  PopNode(Node_type **node_ptr);
      void  Push(Item_type item);
      void  Pop(Item_type *item);
} ;

void  Error(char  *s) {
   printf("%s,\012", s);
}

;
Node_type::Node_type(Item_type item) {
    {
      p->info = item;
      p->next = 0;
   }
}

;
void  Stack_type::PushNode(Node_type *node_ptr) {
   if (node_ptr == 0)
      Error("Attempt to push non-existant node");
   else  {
      node_ptr->next = top;
      top = node_ptr;
   }
}

;
void  Stack_type::PopNode(Node_type **node_ptr) {
   if (top == 0)
      Error("Trying to pop from empty stack");
   else  {
      *node_ptr = top;
      top = *node_ptr->next;
   }
}

;
void  Stack_type::Push(Item_type item) {
   PushNode(MakeNode(item));
}

;
void  Stack_type::Pop(Item_type *item) {
   Node_type *node_ptr;
   PopNode(&node_ptr);
   *item = node_ptr->info;
   free(node_ptr);
}

;
```

- Remarks

This program is from Kruse *et al* [11].

There are two errors in the converted program:

In Node_type::Node_type(), the local variable ‘p’ is still referenced. This is because this section of tree was the ‘else’ part of a pattern to find an if-else statement, and it did not have the MFTB if-replace-specs applied to it.

In Push(), the call to MakeNode has not been updated. This is because the result of the call is not assigned to a variable, and no pattern exists for such calls of an MFTB.

### 6.12.3. Vectors

- Input program

```
// #include <stdio.h>

#define NUM_ELEMS  1000

int heap[NUM_ELEMS];
int heapPtr;

typedef struct vec_tag {
      int first;
      int length;
}  vector;


void start() {
   heapPtr = 0;
}

int size(vector v) {
   return v.length;
}

vector newvec(int s) {
   vector newV;
   if (s >= 1) {
      newV.first = heapPtr;
      newV.length = s;
      heapPtr += s;
   }
   return newV;
}

int index(vector v, int i) {
   if ( (1 <= i) && (i <= size(v)))
      return heap[v.first + i - 1];
   else
      return 0;
}

void alter(vector &v, int i, int n) {
   if ( (1<=i) && (i <= size(v)) )
```

```
      heap[v.first +i-1] = n;
}
```

- User input

  CTB: vec_tag
  MFTBs: size, index, alter

- Converted program

```
int  heap[1000];
int  heapPtr;
class vector {
   private:
      int  first;
      int  length;
   public:
      int  size()const  ;
      int  index(int  i)const  ;
      void  alter(int  i, int  n);
} ;

void  start() {
   heapPtr = 0;
}

int  vector::size()const   {
   return length;
}

vector newvec(int  s) {
   vector newV;
   if (s >= 1)  {
      newV.first = heapPtr;
      newV.length = s;
      heapPtr += s;
   }
   return newV;
}

int  index(vector v, int  i) {
   if (1 <= i && i <= size(v))
      return heap[v.first + i - 1];
   else
      return 0;
}

void  vector::alter(int  i, int  n) {
   if (1 <= i && i <= size())
      heap[first + i - 1] = n;
}
```

- Remarks

This example is from a student exercise written by the author.

For an unknown reason, ctrans could not convert newvec, and so this was not made an MFTB.

No pattern exists to convert a function such as start().

(Correctly) ctrans prints an error message and does not convert index(). This is because the call to size() could modify the argument ‘v’. However, it does create a header for this in the class definition.

size is correctly converted into a ‘const’ function.

### 6.12.4. List

- Input program

```
typedef int ItemType;

int isEmpty(char * s) {
   return (s[0] == 0);
}

struct ListNode {
      ItemType item;
      struct ListNode *next;
};

struct ListTag {
      struct ListNode *first;
};

typedef struct ListTag List;


struct ListNode *MakeNode(ItemType it) {
   struct ListNode *temp;
   if ((temp = malloc(sizeof(struct ListNode))) == 0)
      return (struct ListNode *) 0;
   temp->item = it;
   temp->next = 0;
   return temp;
}

int isEmpty(List l) {
   return (l.first == 0);
}

int length(List l) {
   int i = 0;
   while(isEmpty(l)) {
      l.first = l.first->next;
      ++i;
   }
   return i;
}

ItemType head(List& l) {
   if (isEmpty(l))
      error("attempt to remove head of empty list");
   else {
      struct ListNode *ln = l.first;
      l.first = l.first->next;
      return ln->item;
```

```
   }
}

void add(List& l, ItemType it) {
   struct ListNode *ln = MakeNode(it);
   struct ListNode *temp;
   temp = l.first;
   while(temp->next != 0)
      temp = temp->next;
   temp->next = ln;
}
```

- User input

  conversion I:
  CTB: ListNode
  MFTBs: MakeNode

  conversion II:
  CTB: ListTag
  MFTBs: isEmpty [second occurrence], length, head, add

- Converted program

```
typedef  int  ItemType;

int  isEmpty(char  *s) {
   return s[0] == 0;
}

class ListNode {
   private:
      ItemType item;
      struct ListNode *next;
   public:
      ListNode(ItemType it);
} ;

class List {
   private:
      struct ListNode *first;
   public:
      int  isEmpty()const  ;
      int  length()const  ;
      ItemType head();
      void  add(ItemType it);
} ;

ListNode::ListNode(ItemType it) {
   item = it;
   next = 0;
}

int  List::isEmpty()const   {
   return first == 0;
}

int  length(List l) {
   int  i = 0;
```

```
   while (isEmpty(l))  {
      l.first = l.first->next;
      ++i;
   }
   return i;
}

ItemType List::head() {
   if (isEmpty())
      error("attempt to remove head of empty list");
   else  {
      ListNode *ln = first;
      first = first->next;
      return ln->item;
   }
}

void  List::add(ItemType it) {
   ListNode *ln =  new ListNode it;
   ListNode *temp;
   temp = first;
   while (temp->next != 0)
      temp = temp->next;
   temp->next = ln;
}
```

- Remarks

This example is somewhat contrived, and was written to demonstrate some of the capabilities of ctrans.

The ListNode constructor is converted perfectly here.

In add(), the call to the ListNode constructor has been converted correctly, but ‘new’ statements are not printed correctly by the code output module.

ctrans has correctly refused to convert length, because of the isEmpty() call, but has still added a prototype for it to the class definition.

# 7. CONCLUSIONS

## *7.1. General*

The pattern matching engine of ctrans is fully implemented and has been fairly thoroughly tested. This is the heart of the project, and where most of the design and implementation effort has been directed. It has been found, partly through the successful patterns implemented, and partly through the effects of patterns with bugs in them, that the engine will reliably apply the exact semantics represented in any pattern.

Ctrans as a whole has been found to produce perfect conversions for certain example programs, and reasonable conversions for all other programs. Several problems with the transformations have been shown by the tests performed; however, most of these are due to either inconsistencies in the action of Cppp or the incomplete nature of the code output function. Some problems are due to the patterns, and would have to be addressed by modifying these. However, the set of patterns written covers a very wide range of possible input programs, and should be capable of converting large, commercial C programs with only minor additions.

For many of the conversions that ctrans is not capable of, for example MFTBs with more than one CTB argument, insufficient analysis has been performed to really know what the conversion should be.

The C++ code output from the program is again very good for the example programs, but the procedures that print a syntax tree as code are known to be lacking in certain areas. There are a few syntax tree nodes that no printing procedure has been written for, and there are also cases where the current output functions will produce syntactically incorrect code.

Ctrans' user interface is quite reasonable, given that it was a low priority on the project. It is purely a CLI interface, and it would benefit from more checks on the users input, but it has enough sophistication to, for example, recognise "q" as the quit command but ignore "quite". Of course, a more complete package would have features such as an extensive Unix manual page, and better on-line help facilities than a list of the available commands (which is what ctrans has currently).

The tool is almost at a stage where an evaluation of the overall concept of adding ADT-like features to a program could be performed. To do this, a slightly more robust code output would be needed, and probably a few more patterns to cope with a wider range of C features. Then C programs with a more genuine purpose could be found and converted, and the output analysed for extendibility and clarity. Even if it was found that very little of these programs

could be converted, it seems quite possible that the conclusion would be having part of a program object oriented is better than none at all.

However, ctrans is very close to being useful now, especially for the casual programmer who want to add classes to a basic program, and already knows what those classes will be. It will convert a program quickly and efficiently, without missing any sections that need changing. Some aspects need refining, for example the code output, but these refinements would not take long.

There are also several functionalities missing. These are described in sections 7.5.1 and 7.5.2, but briefly they are:

- no Type II or III transformations
- no user setting of protect specifications
- no detection of required minimum protect specification
- no user choice for actual name of CTB
- no facility to add more member functions to an existing C++ class

Some of these are purely cosmetic, but the Type II/III conversions, and automatic determination of protect specifications, are features which would have been beneficial to the output program structure (in some cases). The automatic protection setting would have enabled the conversion to take maximum advantage of the data hiding potential in the object oriented paradigm. Adding functions to a class would have been a useful convenience, but Type II and III conversions are of debatable value - such conversions may be very useful, or may never be needed. The other work on object-oriented transformations discussed in section 2.3 makes no mention of such conversions.

The application could also be extended quite easily to convert programs on a large scale, which has been the objective of the tools created by other researchers (section 2.3). While it is possible that ctrans will be slow with very large programs, the conversion of structs it carries out are very similar on a code level to the ones performed by the tool of Gall *et al* [3]. This tool and similar ones have sophisticated facilities for aiding the restructuring of a program, but a human expert is still required. If a new program structure was created by such an expert, ctrans could be used to produce code that is as "object-oriented" as the output from such tools. Also with the extension to Type II conversions, the ability to create one-off structure objects may enable ctrans to produce better OO designs.

Finally, there has been a second and unforeseen part to this project, in that a new language for the specification of program transformations has been created, together with a 'compiler' (pt) for this. Overall, it is felt that this is a very good idea, but it is still in a developmental stage. The pattern semantics described above need some more refinement before they describe a fully flexible and clear language. Some of the problems with the language are described in section 7.3, and a few possible improvements can be found in section 7.5.

The pt program, for converting patterns into C++ objects, definitely need some work to improve especially the error checking performed on the input

file. However, it is intelligent enough to fill in all of the utility components for the nodes it outputs, and leaves the pattern writer free to concentrate on the semantically relevant components.

### *7.2. Complexity of C*

The complexity of the C programming language has been a major contributor to the list of obstacles faced in the project, and so a short description of some of the language based factors that have caused problems is given below.

Firstly, the different ways that exist of declaring a struct was a consistent cause for concern during the analysis phase, and has added some complexity to the resulting patterns. With the pattern matching algorithm, this has not caused as many problems as had been feared, but there still have to be two separate patterns for converting structs. It is also unfortunate that the struct tag is not a new type in C (which is a problem that has been rectified in C++), as this means that all references to "struct CTBtag" must be updated.

Typedefs also add problems, as whenever a typedef-name is used, whether of not it refers to the CTB has to be known. However, declaring type synonyms is possible in many languages, just more common in C.

Also common in C is the extensive use of pointers. This means that most checks have to be made twice; once for a non-pointer variable, and once for a pointer variable. Although this would be true for other languages that make use of pointers, the C++ address operator (&) means that extra checks were sometimes required.

Two other peculiarities of C are the pre-processor command system, and the facilities for creating interfaces across files. These did not cause any difficulties for this project, as they were both ignored, but may be problematic if, for example, a commercial version of this tool was developed.

Finally, possibly the biggest problem with C was its size. This meant that a lot of time was spent in familiarisation with the C grammars and syntax trees encountered, and also that there are usually several different ways of creating a C statement to have a given effect. C is also a fairly old language, with some slightly unusual syntax rules, and so (as predicted) many new syntax tree patterns were found as the project progressed.

Examples of these are when a class object declaration and call to its constructor are combined, as shown below.

*ClassName var-name*(*args-for-constructor*);

Also, when a function returning a pointer is defined, the indication that the function returns a pointer is not in the node for the function return type, but in the node for the function declaration. The implicit bracketing is therefore as below,

and although this is expected for C syntax, it again necessitates the use of additional patterns.

int (**funcName*(*arg - decl - list*))

and not

(int*) *funcName*(*arg - decl - list*)

Finally, there are still several types of node in the syntax tree produced by Cppp which are always empty for the trees of the programs studied so far, and it is not known what these nodes would represent if they were not empty.

## *7.3. Pattern matching*

### 7.3.1. Overall evaluation

The chief alternative to pattern matching was an intelligent, self-converting syntax tree. It is difficult to evaluate the pattern matching approach without also having experience of this alternative approach.

Overall, it seems that pattern matching is an excellent basic idea, being both descriptive and intuitive, but the final system has become very complicated. A possible argument is that the concept is fundamentally inadequate for describing the conversions that are required, and the many extensions to the matching algorithm are really just hacks that can not fix this. Certainly the extensions, such as the number-to-find, and conditional trans-specs, detract from the clarity of the original idea.

One of the key reasons for using pattern matching was that when new problems with conversions were encountered, it should only be necessary to write a new pattern to cope with these. In fact, it has been necessary to change the pattern matching semantics about as often as just writing a new pattern has been sufficient. The same trend can be seen in the list of work left to be done in section 7.5.

The counter argument to this is that firstly, the conversions **are** very complicated, and the pattern matching algorithm has to reflect this. Secondly, the pattern language has been improving all the time, and it is now powerful enough that it will be able to cope with almost any new patterns that are found. Evidence for this are the two most recent ideas for patterns, neither of which have required a change in the pattern semantics. The first of these was required to convert MFTBs that return a pointer, but not to a CTB. For these functions, the conversion of the function body is exactly the same as it would be if it did not return a pointer. However, a PtrDeclarator will be declared (with a FctDeclarator as a son) instead of a FctDeclarator, and this will not match the basic pattern for functions.

The pattern semantics are powerful enough to provide two possible solutions to this, the first being the obvious one of duplicating the patterns for the function body in a new pattern, with a different pattern for the function header. The other approach, and the one actually implemented, is to use a TransSpec attached to the node that could be either a PtrDeclarator or a FctDeclarator in the syntax tree. This would have an if-replace-spec to find exactly one FctDeclarator node, and so it does not matter if this FctDeclarator is this node or its son. The pattern is shown in appendix B2.

The second recent pattern idea (not implemented) is for MFTBs that return a CTB (and so will become constructor functions, which do not have return statements) and have more than one return statement. This idea is to replace each return with a goto statement, and create the label for this statement at the end of the function body. The implementation of this would not require any changes to the pattern language.

Another criticism of pattern matching concerns the syntax needed to describe transformations. The key advantage of pattern matching is supposedly the ease with which the program can be extended to cope with newly discovered foibles of C. This would indeed be a valuable property, as throughout the project, new code patterns that need to be transformed have been discovered. However, it is arguable that it would have been just as easy to change the code for a few procedures in an intelligent tree version of the program, as it has been to re-write the patterns. This is because of the complexity of the syntax of the patterns, which means that a simple seeming pattern may contain detailed requirements that a syntax tree must satisfy in order to match it. For example, an if-find node with only a name specifies that the equivalent node in the syntax tree cannot have any sons. The author has on occasion been confused as to the effect that a certain pattern would have, and that is with the benefit of a long exposure to pattern syntax, and of being the developer of the language.

However, while sometimes it has been difficult to follow the functionality of a pattern, this is to be expected for any new and unfamiliar language, and especially one subject to frequent changes. The complexity of pattern writing is due to the expressiveness of the patterns; more experience with the language and a more stable syntax would enable patterns to be written far faster. Essentially, the patterns describe the transformations of the syntax tree at a considerably higher level than C++ code would. They do not, for example, require an understanding of the data structures of the whole program, or the function call sequence.

Also, the pattern language is very new, and many features have potential for improvement (and many features have been gradually improved throughout the project). If the syntax is not perfect now, that is certainly no reason to consider pattern matching a bad idea

A final problem with using a pattern matching language is that it has been time consuming to develop, evaluate and learn. The intelligent tree approach would have been more straightforward, and could have led to a more complete

application if it had been used. As has been mentioned previously (section 5.4.1), as well as the effort required for the development of the pattern matching engine, writing the actual patterns was a difficult and lengthy process.

A key indication of the success of pattern matching is the fact that many of the limitations of the program are due to the patterns it uses. This means that the program code would not have to be changed to enable it to perform conversions on a wider range of programs, just the patterns. In combination with possible alterations to the sMatch and sCreate functions for dealing with special macros, new patterns should be able to cope with any conversion of tree nodes that is required.

This is the real value of pattern matching, and it adds a great deal to the usefulness of the program that it can be extended to cope with unexpected language constructs.

### 7.3.2. Pattern language syntax

There are two specific points about the syntax of pattern matching that seem very awkward, and militate against the whole concept. These are that managing lists is very hard, and that only one pattern can apply to each section of code. The latter will be dealt with first, and the problems stem form the fact that once an if-find pattern has matched a section of tree, that pattern is the only one applied to it. In particular, star-nodes within this pattern can only be modified by sub-trans-specs of the if-find that matches.

This means that if a conversion is to be applied to all code in the source program, for example to convert "*var* = *var* + 1" to "++*var*", it must be embedded in all sections of all patterns that cover function bodies. In other words, all of the patterns converting functions would have to be changed in exactly the same way, which is a tedious process that is prone to errors. Also, to add a new section to a pattern, an in depth understanding of the other transformations performed by that pattern is needed, to avoid duplicating or obstructing its functions.

However, if the whole of a pattern did not have to apply, the advantages of representing global data discussed in section 4.4 would be lost. Also, there are simple solutions to the problems discussed above. Adding conversions such as creating "++*var*" statements can be achieved easily by having a stage that performs a pre-pass over the input tree, using a separate trans-spec consisting of small patterns which will always apply. The problem of modifications having to be performed on many patterns, and needing an understanding of the whole pattern to make changes, can be solved by re-using certain sections of patterns. Both of these changes are described in section 7.5.5 below, and if they were implemented, the ability to separate patterns especially would eradicate many of the non-optimal features of the pattern matching approach.

The second indication that the pattern language is not ideal is that it is not easy to deal with list nodes (nodes that have a variable number of sons) with the current syntax. A symptom of this is the need for the most recent extension, which specifies that if a star node in the if-replace pattern has sons, these are to be added to the star-node after the sons that it already has.

The other symptom of this that has been discovered is the problem with finding the declaration of the variable that is returned in a CTB - returning MFTB. If this declaration is one declarator in a list of declarators, as shown below, it is not possible to find it.

*CTB-name  returned-var, other-var1, other-var2;*

The if-find pattern that would usually be used to find this is shown in figure 7.1, but it fails here because of the EqualTo macro that checks if it is the same as the returned variable. This will invoke a re-scan as the return statement will occur after this declaration. However, the re-scan only applies to the immediate trans-spec (#2), and so only the DeclaratorList will be searched again. No return statement will be found, and a fatal error will result.

### 7.3.3. Technical drawbacks

An important drawback of pattern matching was found while testing ctrans. This is that if a pattern fails to apply correctly, it is often difficult to find where the problem is, as it could be with the pattern, in ctrans, or in pt.

There are also drawbacks to adding more and more patterns. This is because, for each node in the abstract syntax tree, as every single if-find pattern in the current trans-spec will be matched against it. This would not be a problem in the intelligent tree, as additional checks added to it would only be made on the relevant node. Ctrans has been quite fast with small input programs, a fast computer and not many patterns, but at some point the effect of extra patterns will make it noticeably slower.

Also, although patterns are always declared as 'const' objects, they are quite complex data structures.  Compilers can often avoid allocating memory for constant variables, but for the patterns in ctrans it is suspected that they must be placed in memory, so there is also a run time storage overhead of having extra patterns.

However, none of these are major problems. The difficulty of locating errors would be reduced firstly by thoroughly testing ctrans and pt, to minimise the possibility of the error being in them, and secondly with more experience of the pattern language it would be far easier to see where certain kinds of problems are likely to be caused. The speed of ctrans is mostly determined by the speed of Cppp - the pattern matching engine is very fast in comparison to this, and so a faster algorithm would make little difference. Finally, the memory use of patterns would

be reduced considerably by re-using sections of patterns, as described above and in section 7.5.5.

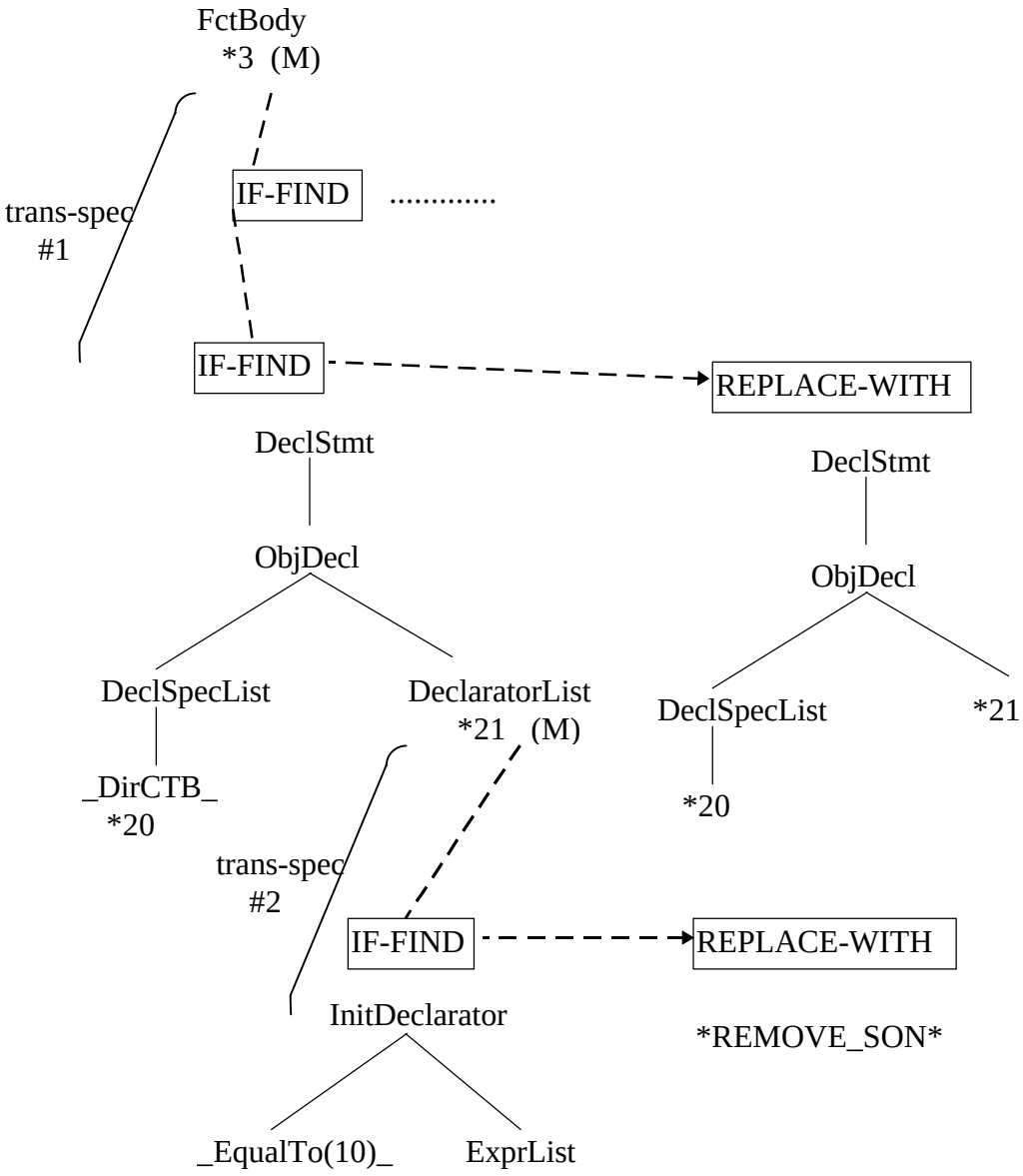



**Figure 7.1. Problem with finding declaration of returned variable**

### 7.3.4. Summary

As an overall conclusion, it is felt that pattern matching was a success. Aside for the appealing nature of the idea, this is felt to be the case for two reasons: the complexity of C, and the fact that there is a lot of development potential in the pattern syntax. Also, experience has chown that increased familiarity with the patterns realises many of the potential advantages of the approach, especially with regard to the ease and speed of developing patterns.

## *7.4. Project evaluation*

### 7.4.1. Decisions

This section aims to provide an assessment of the main decisions made in the implementation of ctrans. Five important choices made have been identified, and these are:

- to use, and print the converted program from a syntax tree
- to use Cppp as the front end for the project
- to use the pattern matching algorithm
- to modify the original syntax tree where possible
- to base conversions on the Cppp parse tree, with links to the semantic tree

These will be discussed below.

Firstly, in retrospect there is no doubt that ctrans had to use a syntax tree. Although the comments and layout of the original program are lost, it is a far easier task to pretty print a syntax tree, or to preserve the line and column number of every comment, than it is to convert a program without the aid of a syntax tree. Also, as had been realised at the time of the decision, there would have been no way to produce the required semantic information about the input program without a syntax tree.

The use of Cppp has been a more problematic part of the project. It is a very good package, with many things to recommend it: a logical, clear syntax tree, many access and utility functions, and resolution of any possible parsing ambiguities. It is very well written, with good error messages, and it has been easy to modify.

The key reservation about the decision to use Cppp is the time that had to be spent in getting the application to compile, and on acquiring a knowledge of the basic structure of the package. Also, some of the programs that ctrans has been tested on have revealed a few bugs in Cppp. These are mentioned in section 6.12, and concern the semantic analysis phase rather than the parsing performed.

However, the alternatives to using Cppp were to use g++ or to create a front end based on a Yacc grammar, which would perform semantic analysis as well as parsing (see section 4.2). Given the author's inexperience in the field of compilers, and the nature of the g++ package, the time invested in Cppp is probably no more that would have been spent in producing a workable front end with either of the other two options. Therefore it is felt that the choice of Cppp was justified.

The pattern matching concept is analysed in detail in the previous section, so it will be covered only briefly here. While pattern matching is almost certainly a superior method, the decision to use it is questionable given the limited amount of time that was available to produce an implementation. The project would undoubtedly have benefited form more time for testing and refinement of the application produced, but the amount of time saved by an ‘intelligent tree’ approach is not known. It is possible that such an approach would have taken longer, and it has been proved in software engineering that a better design is always worth while.

The decision to modify rather than duplicate as much of the syntax tree as possible was not really a key choice. Ctrans would have worked in very much the same way if the whole tree had been duplicated, but modifying was certainly a better route. This is because it uses less memory, and probably performs conversions significantly faster than a duplicating version would have.

Finally, the resolution of the problem encountered with Cppp semantic trees will be discussed. The solution was to produce a copy of the parse tree, which has pointers to the required information in the semantic tree. This is not ideal, as two trees must be used, and links between the trees must be created and followed. If a future modification to ctrans were to require any additional semantic information, the program would have to be modified to create new links for more nodes. The best solution would probably have been to modify the construction of the semantic tree in Cppp, so that it contained the required information, but also remained similar enough to the parse tree to perform conversions on it.

However, there would have been extra problems with this approach, and in particular with creating the correct links between nodes in the modified tree. Also the advantages it has over the current solution are only convenience and memory usage; there would be no functional benefits. Therefore, especially in the light of the time constraints of the project, the choice made was certainly correct.

### 7.4.2. Time management

An analysis of the time and effort spent on the different aspects of the project will be given in this section. Six main phases have been identified, and in chronological order these are: finding and analysing example programs; analysing syntax trees and researching tools; compiling Cppp; ctrans design and creating patterns; coding; writing of report. Of these, the most time was spent on writing the report, but this activity will not be included in the following discussion, as it was not part of the implementation as such.

The approximate number of days spent on each phase are given below.

- ctrans design and creating patterns 21
- coding 16
- finding and analysing example programs 15

- compiling Cppp 11
- analysing syntax trees and researching tools 7

The only phase that was completely separate from adjacent phases was the initial analysis of sample programs. For all the other phases, there were periods where another aspect was being considered simultaneously, and so the days spent on each are only a very rough guide. The amount of time spent on the sections seems reasonable in all cases; the implementation was not one where programming could start immediately, and so the research and analysis phases were essential to the overall success of the project.

Starting with the example program analysis, there was a large amount of time spent on this - nearly as much as on the coding. This is because, firstly, it was the first stage of the project and time also had to be spent on finding out about the background to the work. For example, previous work and compiler techniques had to be investigated. Secondly, the phase was lengthy because of the difficulties in finding sample programs, as outlined in section 3.1.

The information gathered about the tasks ctrans would have to perform during this phase was absolutely necessary. Without it, only the most obvious conversions would be known, and there is no way that these would be adequate for a real C program. The only knowledge that was not used directly in the implementation is the Type II and III conversion procedures, but these proved useful later for evaluating the flexibility of the pattern matching approach. They are a new area for future work to examine.

The time spent on analysing syntax trees and researching tools was also necessary for the implementation to proceed. It was perhaps unprofitable to devote time to understanding the syntax trees produced by C-tree, when this package was not, and could not have been, used in the implementation (as it only parses C, not C++). However, knowledge of these syntax trees provided a basis for evaluating the Cppp syntax trees, which were considerably easier to understand.

The next phase, during which the compilation problems of Cppp were resolved, had no positive impact on the final application, and so could be seen as time wasted. However, if the time had not been spent on getting Cppp to compile, then a different front end would have had to be used. This would almost certainly have required even more time to develop than was spent on Cppp, and so the time invested in compiling Cppp is seen as being justified.

For the next two phases, it can be seen that a long time was spent on design, and less on the actual programming. This is the correct approach for a large software project; however, the time restrictions imposed that ctrans was produced under meant that a full testing and evaluation phase was not possible.

The design produced is however structured so that it can be easily extended, and an amount of the design time was also spent on designing patterns.

For the coding, approximately four-fifths of the time was spent on coding ctrans, and the rest was split between the pt tool (approximately 2 days) and the code output module for Cppp (approximately 1 day).

Finally, it should be pointed out that the ctrans design and pattern writing phases have been grouped together. Chronologically, first two or three trial patterns were developed, then the design of the pattern matching engine was created, and after that the patterns were refined and extended. This order was a mistake, as the final versions of the patterns produced necessitated several changes to the pattern matching engine. If all the patterns had been planed in detail in advance, then a slightly more elegant pattern language may have resulted. However, this is easy to see in retrospect, and at the time it was not even fully appreciated that the pattern matching concept would require something a sophisticated as a ‘pattern language’.

## *7.5. Future work*

### 7.5.1. Updates for patterns

As mentioned above in section 7.3.1, it would be good to find a pattern that is able to find the declaration of a returned variable even if it is declared on a list, but this probably need modifications to the program. However, the MFTB pattern for returning CTBs also fails to find this declaration if it has an initialiser, as shown below.

*CTB-name returned-var* = { *initial-values-for-members* };

It should be possible to write a pattern to find this, and produce extra statements to assign the members of the new class the appropriate values found in the initialiser list.

Another pattern that needs some corrections is the pattern for MFTBs returning a pointer to the CTB. In these patterns, any if-else statement that has an assignment to the CTB variable in the test expression is replaced by the else part. However, this else part may contain MFTB calls, or accesses to data members, and so the if-replace-specs for the FctBody should be applied recursively to the else-statement. An example of the errors produced by this omission can be seen in section 6.12.

### 7.5.2. Updates for program

A basic error in the program is that it is only able to create a tree for part of a program, if it has already parsed one file. The reason for this is not known, and it is possibly due to the memory allocation performed by Cppp.

Also the tool reports errors when parsing the standard g++ "iostream.h" library header file, which is due to the Cppp sections of the package, but needs to be corrected.

Another Cppp related problem is that the semantic analysis functions seem to not be performed on certain nodes. The example of this found is shown in section 6.11, where the SimpleName inside a call statement has no links to the object it represents. It would not be easy to fix this problem.

The main error known for the pattern matching engine is for the calling of MFTBs returning a CTB. These functions are recognised by examining the type of the variable that they are assigned to, and if this type is the CTB, the function is presumed to return a CTB variable. However, if they are called but not assigned to any variable, then they will not be converted to member functions. This could be fixed by defining another special macro that performed a check on the semantic tree object for the function, found via the function identifier node. Then macros could be used to determine if the function was an MFTB returning a CTB, or an MFTB returning a CTB *, not an MFTB.

A considerable barrier to the program having full functionality is the module for printing the syntax tree as code. This needs to be extended to cope with all possible tree nodes, which is relatively straightforward, and the functions producing output code with syntax errors need to be changed. The latter may not be easy, as it probably requires analysis on a semantic level. Also the syntax tree created by Cppp should be modified to distinguish, for example, the different source code structures that are represented as a ClassSpec node. This could be replaced by StructSpec, EnumSpec and ClassSpec, which should be a simple modification.

### 7.5.3. Additional patterns

There is a potential for many more patterns to be included in ctrans, and a list of omitted patterns is given below, but testing on more C code would undoubtedly reveal even more.

One probably straightforward pattern that was omitted was an MFTB returning a CTB variable by reference, which is possible but does not seem to be encountered often.

Another possibility that should be allowed for is a combined struct definition and typedef for pointers to that struct, as shown in the example below.

```
typedef struct stack_tag {
    Node_type  * top;
}    * stack_pointer_type;
```

Also missing from the current list of ctrans patterns is a pattern to match an existing C++ class, which would allow more member functions to be added to such a class.

Part of the pattern for MFTBs returning a CTB * variable is to remove allocation of memory for that variable; therefore, there should be in other member function patterns, a sub-pattern to remove statements de-allocating the memory for a CTB * variable.

As well as the definition of functions, it is possible that prototypes for these functions can be declared in a C program. If a prototype is declared for an MFTB, then this should be converted as well as the calls and definition of it. Possibly, the pattern should have the action of removing the prototype completely, as the function will be declared in the definition for the new class.

Ideally, although it will require more analysis of sample programs, patterns to convert functions with more than one CTB as an argument should be produced.

More advanced ideas could be to convert unions to classes making use of inheritance, and to write patterns for template classes and functions. Also the syntax tree nodes that are understood badly can be experimented with, for example having KeyDeclSpec nodes as "const" or "volatile", or having non-empty ExprLists (for initialisation) in function arguments and CTB variable declarations.

### 7.5.4. Types II and III

While an extension to allow Type II and III conversions would involve modifications to most of the peripheral parts of ctrans, it is believed that such an extension would not require any changes to the pattern matching engine. When the choice was made to use the pattern matching algorithm, a trial pattern for a type II transformation was developed, and this was found to be relatively simple.

However, changes would be needed, firstly to the intro-actions that create the lists of functions and structs found in the program. This would have to create a list of all global variables for Type III. It is easy to get a list of all the variables declared within a function from the Object_Function CpppAst node. The tree manager and interface will have to ask the user for, and store, which variables make up the CTB. Also the new signatures of functions will have to be stored in the MemFn class.

The special match and create functions will have to be modified to access this information, so matches such as _VarToBeMem_ will be used, and creates such as _ClassVarEquiv(arg)_ to find the member variable of the new class equivalent to the argument identifier.

### 7.5.5. Extensions to programs

Additional features for the two programs, pt and ctrans, that should prove useful are described in this section.

Starting with the pattern transformer (pt), this would benefit greatly from more syntax checks on the input pattern file. Currently, it is capable of detecting incorrect special macros, and complains if a replace-with node has been left out. A facility to record the line number in the input file at which any problems occurred was found to be invaluable.

However, there are several key things that are not checked. It does not check that names given for IfFindNodes are valid for Cppp nodes, so a simple spelling mistake can prevent a pattern from being applied when it should. This is not such a problem for ReplaceNodes, as the C++ compiler will report an error if the name is miss-spelt, but could also be incorporated for these. Also, there are very few checks on the components for a node, and many combinations other than the ones given in the semantic description of nodes in sections 5.3.4 and 5.3.5 will be allowed for nodes.

Also pt currently reads in the list of stars to remove as a string, which allows it to contain non-integers, and these will cause the compilation to fail. Even worse, if there is a space between two of the integers in the list, then the list after the space will be lost, but no error reported. Another more technical update required is a check that the file name given as an argument actually exists (currently the program crashes if it does not).

A change that would be extremely worthwhile for the pattern syntax would be to provide a mechanism for incorporating other patterns into a pattern. This would enable the parts of patterns common to all MFTBs, for example, to be kept in a separate, common file rather that cutting-and pasting (large) sections between pattern files. The advantages would be that modifications to patterns would only have to be performed in one place, extra patterns to be used in, for example, all function definitions would only have to be added in one place, and less memory would be needed for the patterns as parts of them would be re-used.

This could be implemented by allowing the line for an object to specify its name, rather than being given a sequential number. For example, the syntax "#Explicit-irs *irs-name rest-of-if-replace-spec-components*" could be used instead of "#IfReplaceSpec *rest-of-if-replace-spec-components*", and "#Irs-reference *irs-name*" used in patterns to include a specific if-replace-spec.

Finally (although perhaps a modification that should be performed first), the structure of pt was developed rather hastily, and it could probably benefit from a re-think.

Moving on to ctrans, there are a few features which are partially implemented, but lack for example the user interface facilities for them to work. The first of these is concerns the protection specification for members of the new

class. Currently all variables become “private”, and all functions become “public”. Having members as private may well cause compilation errors, as the member could well be accessed from a non-member function, and having members as public means that the maintenance benefits of data hiding are being lost. To prevent this, two new features should be introduced: the ability for the user to select protection specifications explicitly, and intelligent default settings to be used otherwise. To implement the latter, a check would have to be made to find the most restrictive possible protect setting for each member, and have this as the default. This could probably be performed as a separate pass over the syntax tree using the CpppTraverseActions feature, which would check where accesses to member-variables-to-be occur from.

Although ctrans can distinguish between overloaded functions, when the choice of functions to become members is presented to the user, only the function names are shown. This does not allow the user to know which of a set of overloaded functions they are selecting, and so the full function prototype for each potential MFTB rather than just the name should be printed. On a similar note, ctrans stores all the typedef names that refer to each struct, but when listing the structs available for conversion, only prints out the struct-tag for each. As well as listing all of the names found for each struct, ctrans should allow the user to enter explicitly the name they want for the new class.

A mostly aesthetic feature would be to convert the first letter of the name of the new class to upper case, so that it conforms to C++ naming conventions. This would be very easy to implement.

An extremely useful feature would be a pre-pass pattern matching stage, as mentioned in section 7.3. This would apply a special, separate set of patterns to the whole tree, and could be used for targeting the tree to C++. For example, statements such as “*var-name* = *var-name* + 1;” could be replaced with “++*var-name*”. Also, all the C++ keywords that do not exist in C could be found and replaced, with an underscore character pre-pended. For example, the identifier “class” would become “_class”, and “new” would become “_new”. However, this relies on having a C parser to read in the input program, and an input program guaranteed to be in C. Currently, as Cppp is a C++ parser, it would flag any use of a C++ keyword as an identifier as an error.

The next area in which ctrans could be improved is the user interface. The current CLI is acceptable, but could certainly be improved, for example with more help information. Also a tab-completion facility would be useful for reading in files, and more precise error messages could be generated. Finally, the system used for selecting the CTB and MFTBs should be re-evaluated; current HCI opinion is that modal systems are not as clear as single-mode systems.

However, it is felt that a system such as ctrans would be far easier to use if it had a graphical user interface. This would allow the user to see the program code while selecting the CTB and MFTB, rather than having to remember which functions should go with which structs.

Lastly, features that would be required for a commercial version of ctrans will be discussed. As it is, ctrans does not deal with converting programs spread across multiple files, and it is left up to the user to ensure that any conversions performed are consistent. It should be modified to convert struct definitions in a separate file from the functions that will become members, perhaps prompting the user for any file in which MFTBs may be defined. It should also store the new signatures of functions converted to members, so that it is able to modify further files that use these functions.

A point of particular concern, in the light of the aims of the project in terms of program maintainability, is that ctrans discards any comments that the source program may contain. One way of preserving these would be to have the lexer return them as a special token, and store them with the syntax tree node occurring immediately before (or after) them. They could then be output in roughly the correct place, on a separate line, and would at least not be lost. Preserving the pre-processor macros would be very hard, but perhaps not impossible. The macros would have to be expanded before the conversion, to ensure the syntax tree represents the complete program semantics, but they could be stored prior to that. Then, after the conversion, they could be reverse-expanded into the code, by matching any sections that could have been produced by a macro and asking the user if they should be replaced by a macro call. For example, all constant integers with the value ‘0’ would be (subject to user acceptance) replaced by the macro ‘NULL’.

The final point is that for large programs, it is possible that the speed of execution and memory use of ctrans would not be adequate. These were considered during the design, but were not high priorities and could undoubtedly be improved.

### 7.5.6. Advanced extensions

The first more ambitious extension is one that the structure of ctrans would make for easier than might be thought. This is to convert programs in languages other than C, into programs in C++ or in some other OO language.

Parts of ctrans are very dependant on the format of Cppp syntax tree, for example having links in the semantic tree from SimpleName nodes to a unique object that the identifier refers to. However, this layout is *not* restricted to the C or C++ language, and in fact the writers of Cppp have developed a very similar package to parse Object Pascal [17]. This would allow ctrans to be easily updated to take Pascal input, and produce Object Pascal (or C++) output. However, a completely new set of patterns would be required for this, and a new code pretty printer (for Object Pascal).

The next advanced extension is to recognise the struct that should become objects, and the functions that should become members, automatically. This would not be so simple, and would require data-flow analysis of the input

program. Also, for Type II programs, it may be possible to use data flow analysis and some heuristics to guess that a set of two integers is often used, and this would be a good candidate for an object.

The final extension that will be covered is other program transformations. As ctrans is, only the patterns, user interface and special macros are specific to converting non-OO code into OO code. It would be straightforward to write patterns to perform other types of modifications, and update possibly the user interface, special macros and data stored by the tree manager, depending on the complexity of the transformations. At a fairly low level, possibilities that could be investigated are replacing 'goto' statements with control structures, and replacing 'for' loops with 'while' loops. However higher level conversions, along the lines of non-OO to OO, may also exist that would benefit from a tool to partly automate them.

### *7.6 Summary*

A summary of the achievements of this project will now be presented. This will cover ctrans, the pattern matching concept, the pattern language and pt, and finally the value of a tool such as ctrans.

The ctrans program that has been developed is a well structured application, and the main 'pattern matching engine' of the program is fully developed and tested. Unfortunately, due to time constraints, there was no testing and refinement stage for the application as a whole.

The patterns currently used by ctrans are nearly complete. They require only a few refinements to be capable of performing all known Type I conversions. However, the problem of MFTBs with multiple CTB arguments still needs to be addressed.

The pattern matching concept has been a major success of the project, and (as mentioned in section 7.5.6) it has potential uses beyond the conversions of interest here. Even for the program transformations needed for ctrans, the patterns provide a high level and easily understandable way of specifying them.

For pt and the pattern language, there are several ideas for improvements listed in section 7.5.5, and the implementation for all these would be straightforward. With these modifications, the pattern language and 'compiler' would form a very powerful and complete system for specifying program transformations.

Finally, there has been insufficient time to study the application of ctrans on larger and more diverse C programs, and to investigate the benefits that conversions of the type performed by ctrans have for program structure and maintainability. However, there are indications that these conversions are required,

for example Fitzpatrick [8] who converted a C program to C++ manually, and Gall *et al* [3] and Ong and Tsai [7] have also created tools to assist with such transformations. Where such transformations need to be performed, ctrans would certainly be a useful tool to help with it.

## REFERENCES


[1] K. Lano, H. Haughton (1994*), Reverse Engineering and Software Maintenance: A Practical Approach*, McGraw-Hill Book Company, London

[2] I. Jacobson, F. Lindstrom (1991), "Re-engineering of Old Systems to an Object-Oriented Architecture", *Proceedings of Conference on Object-Oriented Programming Systems, Languages and Applications, Phoenix, Arizona*, October, pp. 340-350

[3] H. Gall, R. Klosch, R. Mittermeir (1995), "Object-Oriented Re-Architecturing", *5th European Software Engineering Conference, Sitges, Spain*, September. Proceedings p.499

[4] P.H. Winston (1994), *On To C++*, Addison-Wesley Publishing Company, Reading, Massachusetts

[5] A.K. Onoma, W.T. Tsai, F. Tsunoda, H. Suganuma, S. Subramanian (1995), "Software Maintenance - An Industrial Experience", *J. Software Maintenance*, **7**,5 (Sept./Oct.), pp. 333-375

[6] S.C. Choi, W. Scacchi (1990), "Extracting and Restructuring the Design of Large Systems", *IEEE Software*, **7**, January 1990, pp. 66-71

[7] C.L. Ong, W.T. Tsai (1993), "Class and Object Extraction from Imperative Code", *J. Object-Oriented Programming*, **6**, 1 (March/April), p.58

[8] J. Fitzpatrick (1996), "Case Study: Converting C Programs to C++", *C++ Report*, **8**, 2 (February), p. 40

[9] J.R. Hanly, E.B. Koffman, F.L. Friedman (1993), *Problem Solving and Program Design in C*, Addison-Wesley Publishing Company, Reading, Massachusetts

[10] E.S. Roberts (1995), *The Art and Science of C*, Addison-Wesley Publishing Company, Reading, Massachusetts

[11] R.L. Kruse, B.P. Leung, C.L. Tondo (1991), *Data Structures and Program Design in C*, Prentice-Hall International, London

[12] A.V. Aho, R. Sethi, J.D. Ullman (1986), *Compilers: Principles, Techniques and Tools*, Addison-Wesley Publishing Company, Reading,

[13] J.P. Bennett (1990), *Introduction to Compiling Techniques: A First Course Using ANSI C, LEX and YACC*, McGraw-Hill Book Company, London

[14] C-TREE, version 0.02
S. Flisakowski (1996)
ftp://ftp.cs.wisc.edu/coral/tmp/spf/ctree-0.2.tar.gz

[15] J.A. Roskind (1991)
Grammar for C++, Yacc and Lex files cpp5.y cpp5.l
ftp://ftp.infoseek.com/pub/c++grammar/c++grammar2.0.tar.Z

[16] gcc
The Free Software Foundation
ftp://src.doc.ic.ac.uk/gnu/gcc-2.7.2.1.tar.gz

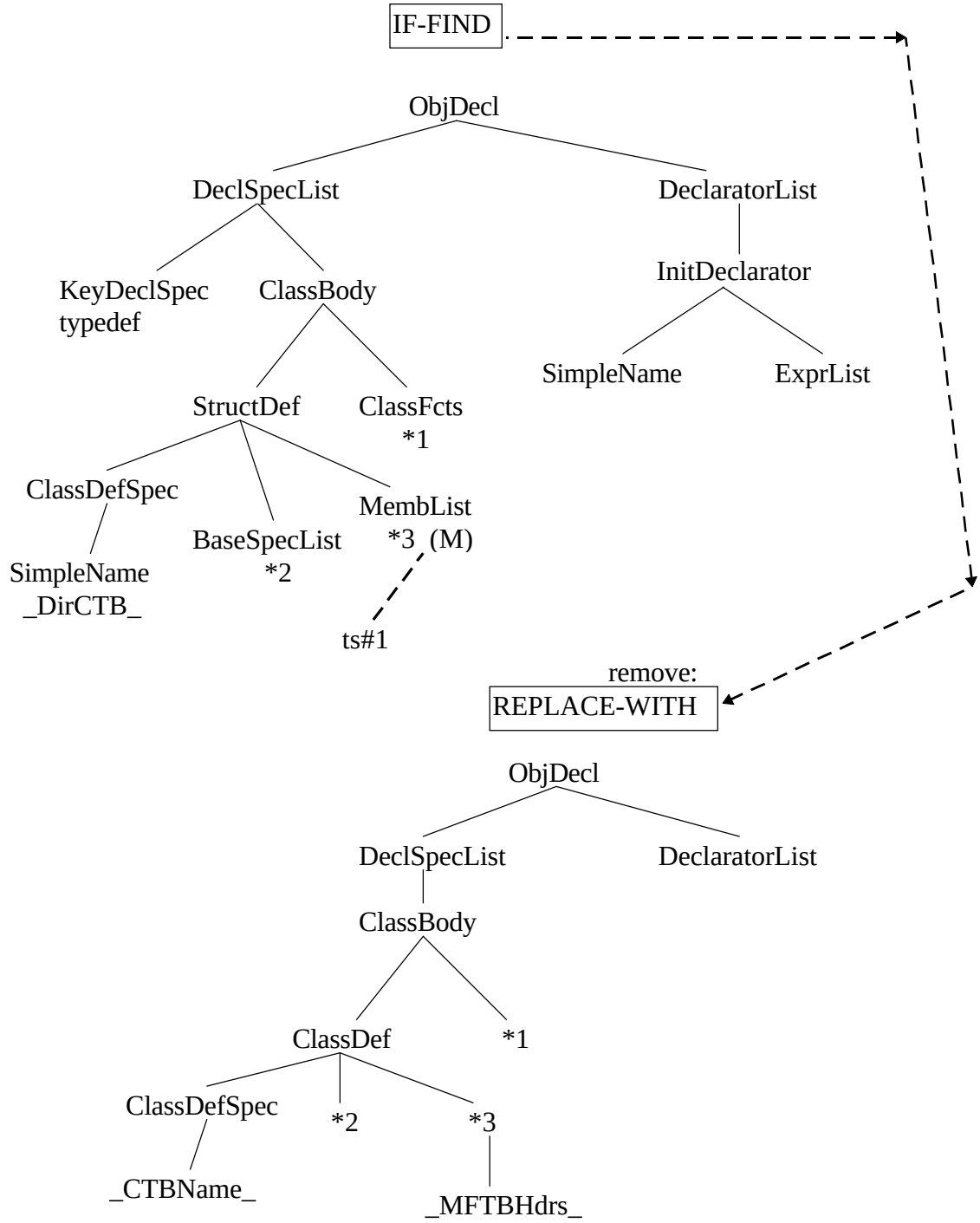
[17] Cppp distribution version 1.83
S. P. Reiss, T. Davis (1995)
file “Organization”
file “oofront.ps”
ftp://wilma.cs.brown.edu:/pub/cppp.tar.Z

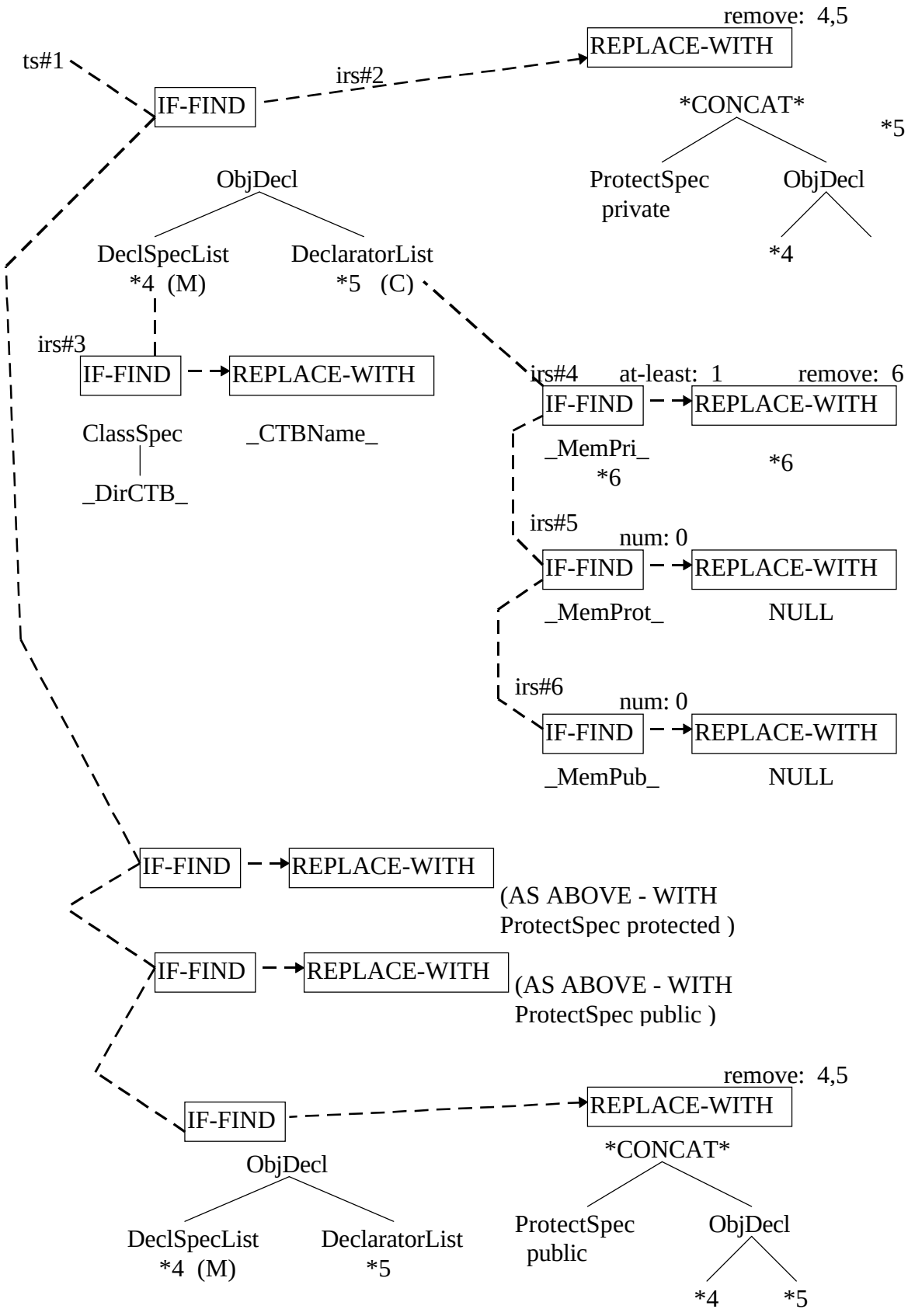
[18] C++ draft standard
The C++ standard committee
(ANSI XJ316, ISO SC22-SG21)
April 28, 1995 version
ftp://research.att.com/dist/c++std/WP

# APPENDIX A

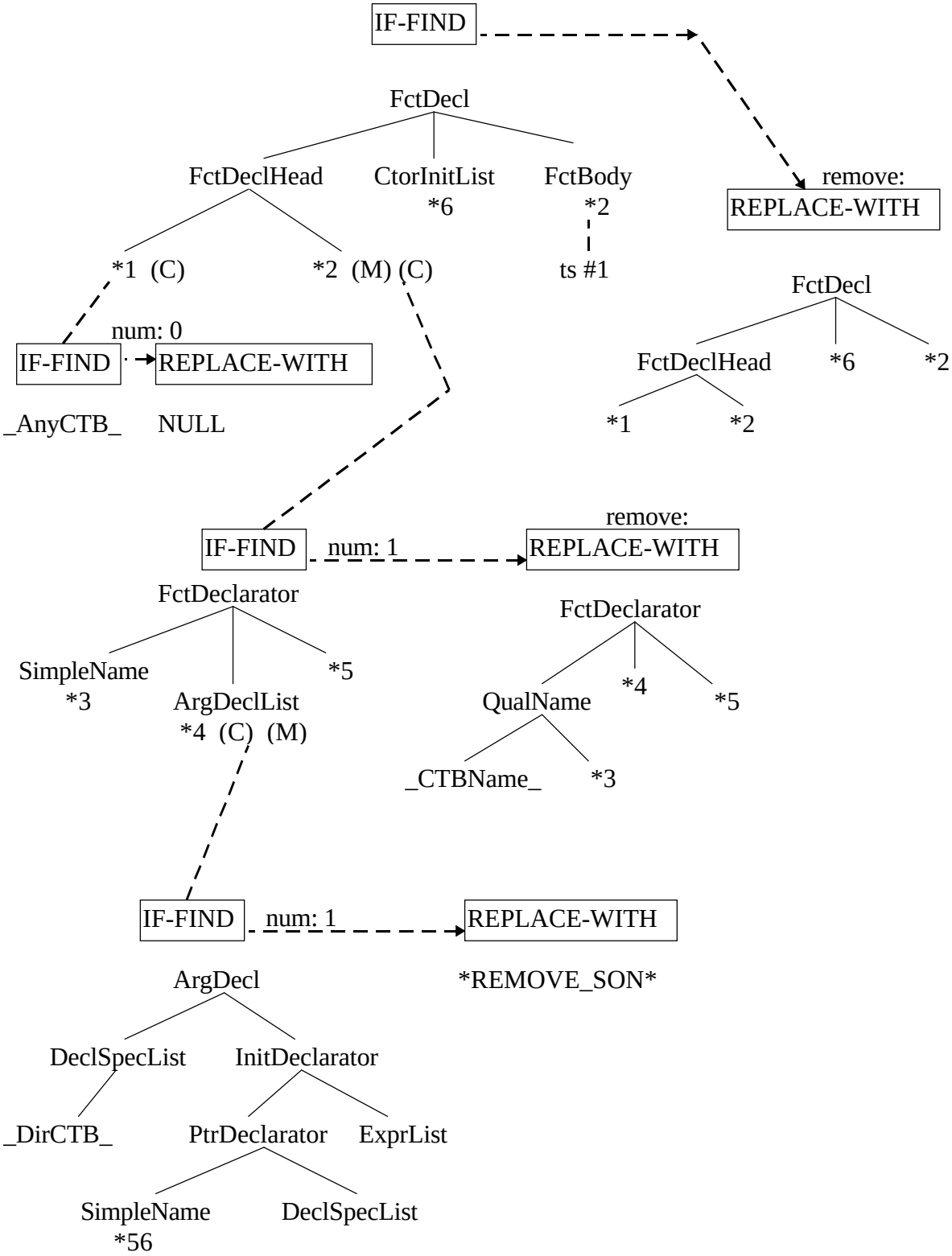
Two graphical examples of the patterns will be presented in this appendix, firstly a complete struct example, and then a representative MFTB example.

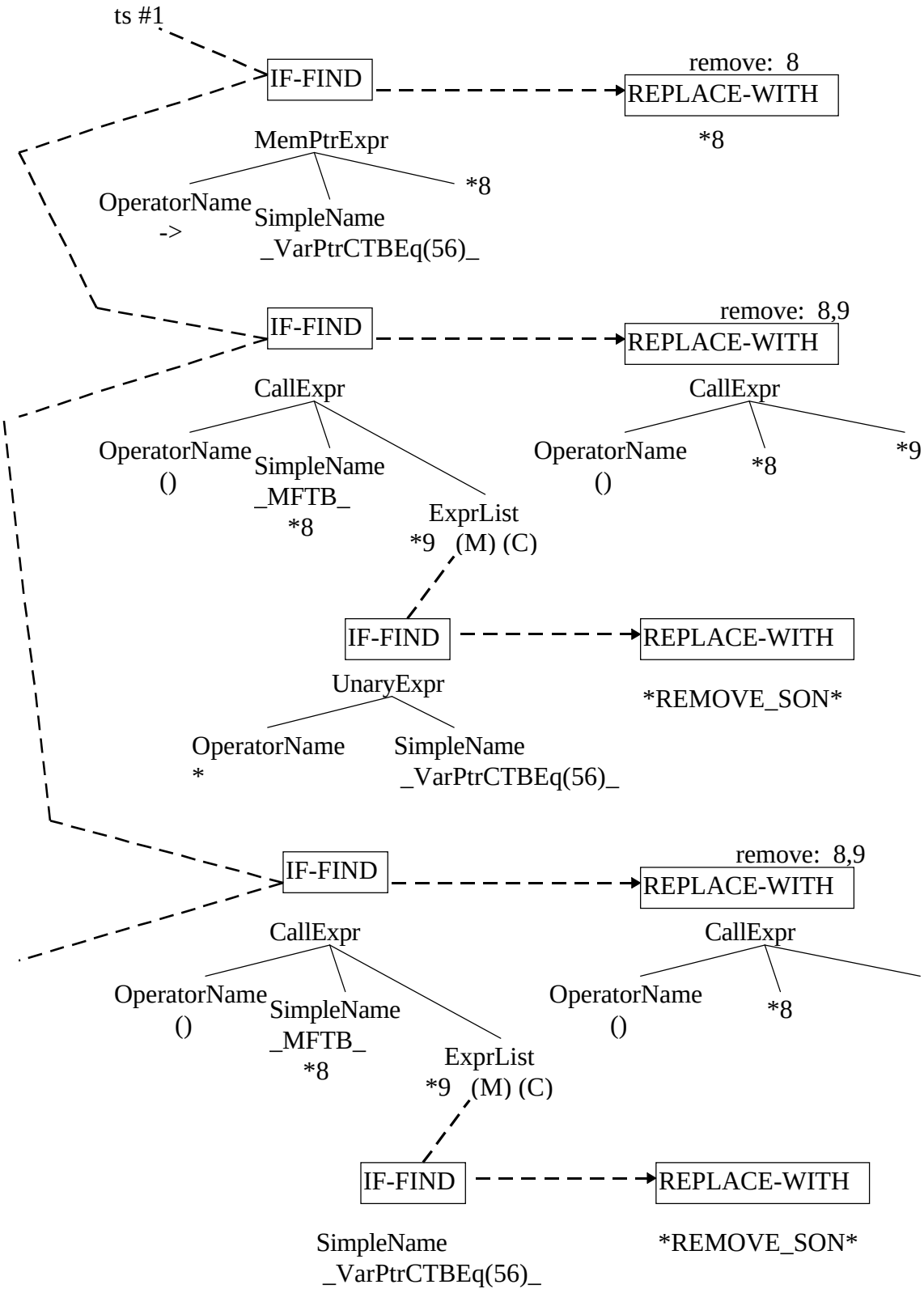
### *A1 Typedef-struct*

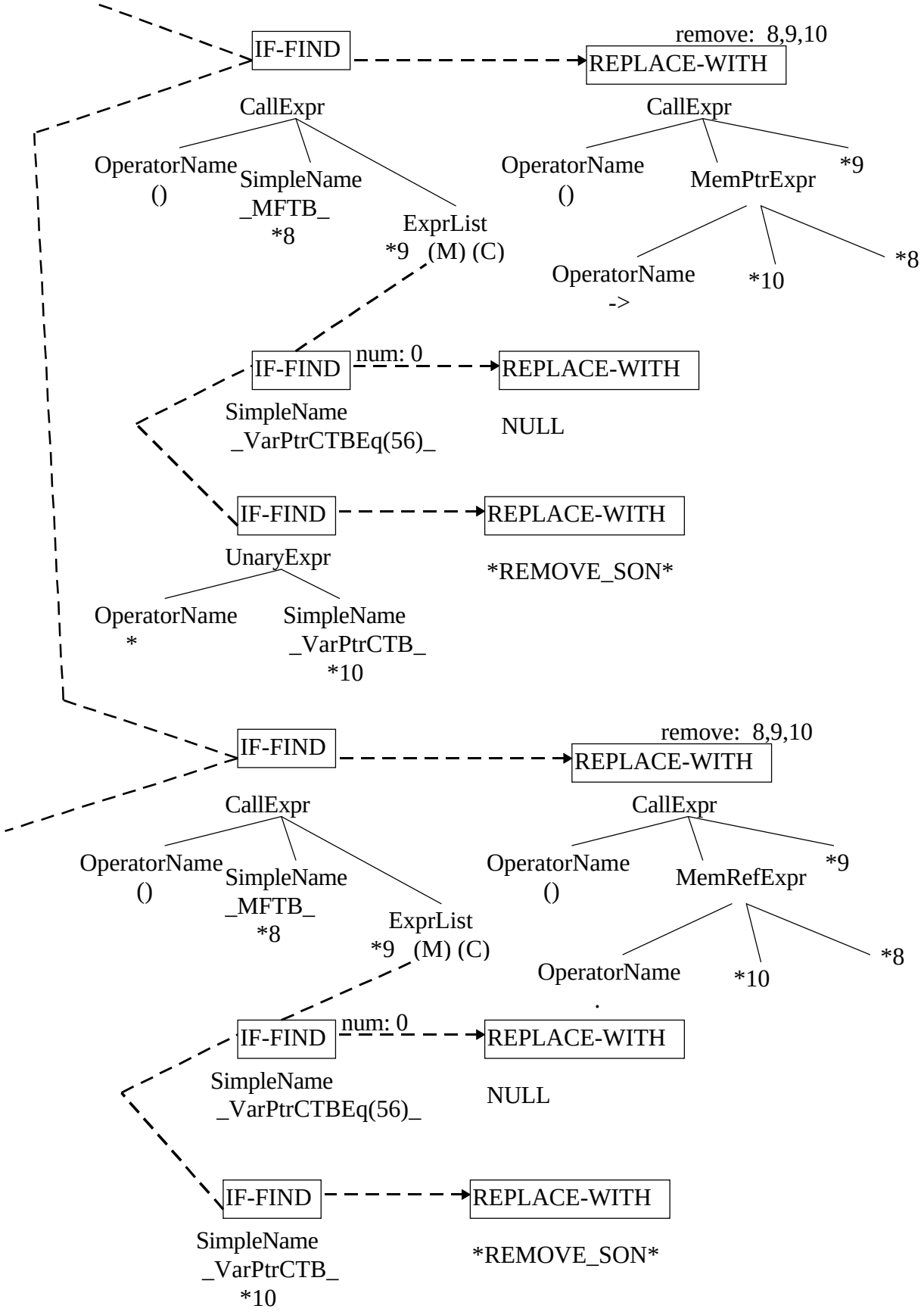



remove: 4,5
REPLACE-WITH
ts#1
irs#2
IF-FIND
*CONCAT*
*5
ObjDecl
ProtectSpec
private
ObjDecl
*4
DeclSpecList
*4 (M)
DeclaratorList
*5 (C)
irs#3
IF-FIND
REPLACE-WITH
irs#4
at-least: 1
remove: 6
IF-FIND
REPLACE-WITH
ClassSpec
_CTBName_
_MemPri_
*6
*6
_DirCTB_
irs#5
num: 0
IF-FIND
REPLACE-WITH
_MemProt_
NULL
irs#6
num: 0
IF-FIND
REPLACE-WITH
_MemPub_
NULL
IF-FIND
REPLACE-WITH
(AS ABOVE - WITH
ProtectSpec protected )
IF-FIND
REPLACE-WITH
(AS ABOVE - WITH
ProtectSpec public )
remove: 4,5
IF-FIND
REPLACE-WITH
*CONCAT*
ObjDecl
ProtectSpec
public
ObjDecl
DeclSpecList
*4 (M)
DeclaratorList
*5
*4
*5

***A2 MFTB-arg-ptr1***




ts #1
IF-FIND
remove: 8
REPLACE-WITH
*8
MemPtrExpr
OperatorName
->
SimpleName
_VarPtrCTBEq(56)_
*8
IF-FIND
remove: 8,9
REPLACE-WITH
CallExpr
OperatorName
()
SimpleName
_MFTB_
*8
ExprList
*9 (M) (C)
CallExpr
OperatorName
()
*8
*9
IF-FIND
REPLACE-WITH
*REMOVE_SON*
UnaryExpr
OperatorName
*
SimpleName
_VarPtrCTBEq(56)_
IF-FIND
remove: 8,9
REPLACE-WITH
CallExpr
OperatorName
()
SimpleName
_MFTB_
*8
ExprList
*9 (M) (C)
CallExpr
OperatorName
()
*8
IF-FIND
REPLACE-WITH
SimpleName
_VarPtrCTBEq(56)_
*REMOVE_SON*


IF-FIND
remove: 8,9,10
REPLACE-WITH
CallExpr
OperatorName
()
SimpleName
_MFTB_
*8
ExprList
*9 (M) (C)
CallExpr
OperatorName
()
MemPtrExpr
*9
OperatorName
->
*10
*8
IF-FIND
num: 0
REPLACE-WITH
SimpleName
_VarPtrCTBEq(56)_
NULL
IF-FIND
REPLACE-WITH
UnaryExpr
*REMOVE_SON*
OperatorName
*
SimpleName
_VarPtrCTB_
*10
IF-FIND
remove: 8,9,10
REPLACE-WITH
CallExpr
OperatorName
()
SimpleName
_MFTB_
*8
ExprList
*9 (M) (C)
CallExpr
OperatorName
()
MemRefExpr
*9
OperatorName
.
*10
*8
IF-FIND
num: 0
REPLACE-WITH
SimpleName
_VarPtrCTBEq(56)_
NULL
IF-FIND
REPLACE-WITH
SimpleName
_VarPtrCTB_
*10
*REMOVE_SON*

# APPENDIX B

This appendix aims to provide a complete guide to the patterns used by ctrans. It is not practical do display all of all of the patterns graphically; a textual representation of every if-replace-spec in every pattern is provided instead.

Names appearing in italics represent code sections or identifiers that will have a special significance in the actual program to be converted. For example, the name *other_ptr_ctb* has been used to represent a variable whose type is pointer to the CTB, and that is not the CTB argument variable of an MFTB. The name *DirCTB* represents a type specifier for the CTB, and *arg-decl-list* represents the argument declarations for a function.

Also a pattern ‘macro’, STD-FUNC, has been used to represent all of the conversions listed below, which occur in every pattern for converting functions.

### *<STD-FUNC>*

*mftb* (----**other_ptr_ctb*----) ⇒ *other_ptr_ctb-> mft*b (--------)

*mftb* (----&*other_dir_ctb*----) ⇒ *other_dir_ctb. mftb* (--------)

*mftb* (----*other_ptr_ctb*----) ⇒ *other_ptr_ctb-> mft*b (--------)

*mftb* (----*other_dir_ctb*----) ⇒ *other_dir_ctb. mftb* (--------)

*DirCTB other_dir_ctb = mftb*(----) ⇒
*CTBName other_dir_ctb = CTBName*(----)

*PtrCTB other_ptr_ctb = mftb*(----) ⇒
*CTBName *other_ptr_ctb* = new *CTBName*(----)

*DirCTB *other_ptr_ctb = mftb*(----) ⇒
*CTBName *other_ptr_ctb* = new *CTBName*(----)

*dir_ctb = mftb*(----) ⇒ *dir_ctb = CTBName*(----)

*ptr_ctb = mftb*(----) ⇒ *ptr_ctb* = new *CTBName*(----)

struct *DirCTB ctb_var1, ctb_var2* ⇒ *CTBName ctb_var1, ctb_var2*

### *B1 Typedef-struct*

typedef struct [*ctb_tag*] {
*members*

} *ctb_type*
⇒
class *CTBName* {
*members*
*MFTB-headers*
}

*members*:
*type-specifier members_all_to_be_public*
⇒
public: *type-specifier members_all_to_be_public*

*type-specifier members_all_to_be_private*
⇒
private: *type-specifier members_all_to_be_private*

*type-specifier members_all_to_be_protected*
⇒
protected: *type-specifier members_all_to_be_protected*

*type-specifier members*
⇒
public: *type-specifier members*

### *B2 struct-only*

struct *ctb_tag* {
*members*
}
⇒
class *CTBName* {
*members*
*MFTB-headers*
}

*members*:
*type-specifier members_all_to_be_public*
⇒
public: *type-specifier members_all_to_be_public*

*type-specifier members_all_to_be_private*
⇒
private: *type-specifier members_all_to_be_private*

*type-specifier members_all_to_be_protected*

⇒

protected: *type-specifier members_all_to_be_protected*

*type-specifier members*

⇒

public: *type-specifier members*

### *B3 typedef-for-CTB*

typedef *DirCTB names* ⇒ /* NULL */

### *B4 typedef-for-ptr-CTB*

typedef *DirCTB *name* ⇒ typedef *CTBName *name*

### *B5 Non-MFTB*

*return-type* [*]*non-mftb*(*arg-decl-list*) {

*fct-body*

}

⇒

*return-type* [*]*non-mftb*(*arg-decl-list*) {

*fct-body*

}

*fct-body*:

<STD-FUNC>

### *B6 MFTB-arg-addr*

*return-type* [*]*mftb*(*arg-decl-list*) {

*fct-body*

}

⇒

*return-type* [*] *CTBName::mftb*(*arg-decl-list*) {

*fct-body*

}

*arg-decl-list:*

*DirCTB &ctb_arg* ⇒ /* NULL */

*fct-body*:

*ctb_arg.member* ⇒ *member*

*mftb*(*args*) ⇒ *mftb*(*args*)
*args*:
&*ctb_arg* ⇒ /* NULL */

*mftb*(*args*) ⇒ *mftb*(*args*)
*args*:
*ctb_arg* ⇒ /* NULL */

<STD-FUNC>

*ctb_arg* ⇒ this

### *B7a MFTB-arg-ptr1*

*return-type* [*]*mftb*(*arg-decl-list*) {
*fct-body*
}
⇒
*return-type* [*] *CTBName::mftb*(*arg-decl-list*) {
*fct-body*
}

*arg-decl-list:*
*DirCTB* **ctb_arg* ⇒ /* NULL */

*fct-body*:
*ctb_arg->member* ⇒ *member*

*mftb*(*args*) ⇒ *mftb*(*args*)
*args*:
**ctb_arg* ⇒ /* NULL */

*mftb*(*args*) ⇒ *mftb*(*args*)
*args*:
*ctb_arg* ⇒ /* NULL */

<STD-FUNC>

*ctb_arg* ⇒ this

### *B7b MFTB-arg-ptr2*

*return-type* [*]*mftb*(*arg-decl-list*) {
*fct-body*
}
⇒

*return-type* [*] *CTBName::mftb*(*arg-decl-list*) {
    *fct-body*
}

*arg-decl-list:*
    *PtrCTB ctb_arg* ⇒ /* NULL */

*fct-body*:
    *ctb_arg->member* ⇒ *member*

    *mftb*(*args*) ⇒ *mftb*(*args*)
        *args*:
            **ctb_arg* ⇒ /* NULL */

    *mftb*(*args*) ⇒ *mftb*(*args*)
        *args*:
            *ctb_arg* ⇒ /* NULL */

    <STD-FUNC>

    *ctb_arg* ⇒ this

### *B8 MFTB-arg-value*

*return-type* [*]*mftb*(*arg-decl-list*) {
    *fct-body*
}
⇒
    *return-type* [*] *CTBName::mftb*(*arg-decl-list*) {
        *fct-body*
    }

*arg-decl-list:*
    *DirCTB ctb_arg* ⇒ /* NULL */

*fct-body*:
    *ctb_arg = anything* ⇒ <error - no conversion>

    *ctb_arg.member = anything* ⇒ <error - no conversion>

    *any-func*(*args*) ⇒ <error - no conversion>
        *args:*
            *ctb_arg.member* ⇒ /* NULL */

    *any-func*(*args*) ⇒ <error - no conversion>
        *args:*
            *ctb_arg* ⇒ /* NULL */

<STD-FUNC>

*ctb_arg* ⇒ this

### *B9a MFTB-return-ptr1*

*PtrCTB mftb*(*arg-decl-list*) {
    *fct-body*
}
⇒
    *CTBName::CTBName*(*arg-decl-list*) {
        *fct-body*
    }

*fct-body:*
    return *return_var* ⇒ /* NULL */

    *PtrCTB return_var* ⇒ /* NULL */

    *DirCTB * return_var* ⇒ /* NULL */

    if ( (*return_var = anything*) == 0)
        *then-stmt*
    else
        *else-stmt*
    ⇒ *else-stmt*

    if ( (*return_var = anything*) == 0)
        *then-stmt*
    ⇒ /* NULL */

    *return_var = anything* ⇒ /* NULL */

    *return_var ->member* ⇒ *member*

    *mftb*(*args*) ⇒ *mftb*(*args*)
        *args*:
            ** return_var* ⇒ /* NULL */

    *mftb*(*args*) ⇒ *mftb*(*args*)
        *args*:
            *return_var* ⇒ /* NULL */

    <STD-FUNC>

    *return_var* ⇒ this

### *B9b MFTB-return-ptr2*

*DirCTB *mftb*(*arg-decl-list*) {
    *fct-body*
}
⇒
    *CTBName::CTBName*(*arg-decl-list*) {
        *fct-body*
    }

*fct-body:*
    return *return_var* ⇒ /* NULL */

    *PtrCTB return_var* ⇒ /* NULL */

    *DirCTB * return_var* ⇒ /* NULL */

    if ( (*return_var = anything*) == 0)
        *then-stmt*
    else
        *else-stmt*
    ⇒ *else-stmt*

    if ( (*return_var = anything*) == 0)
        *then-stmt*
    ⇒ /* NULL */

    *return_var = anything* ⇒ /* NULL */

    *return_var ->member* ⇒ *member*

    *mftb*(*args*) ⇒ *mftb*(*args*)
        *args*:
            ** return_var* ⇒ /* NULL */

    *mftb*(*args*) ⇒ *mftb*(*args*)
        *args*:
            *return_var* ⇒ /* NULL */

    <STD-FUNC>

    *return_var* ⇒ this

### *B10 MFTB-return-value*

*DirCTB mftb*(*arg-decl-list*) {
    *fct-body*
}
⇒
    *CTBName:: CTBName*(*arg-decl-list*) {
        *fct-body*
    }

*fct-body*:
    return *return_var* ⇒ /* NULL */

    DirCTB *return_var* ⇒ /* NULL */

    *return_var.member* ⇒ *member*

    *mftb*(*args*) ⇒ *mftb*(*args*)
        *args*:
            & *return_var* ⇒ /* NULL */

    *mftb*(*args*) ⇒ *mftb*(*args*)
        *args*:
            *return_var* ⇒ /* NULL */

    <STD-FUNC>

    *return_var* ⇒ this

# APPENDIX C

### *CpppAst object heirarchy*

- CpppAstInfo
  - CpppAst_AccessSpec
  - CpppAst_BaseSpec
  - CpppAst_BaseSpecList
  - CpppAst_ClassBody
  - CpppAst_ClassFcts
  - CpppAst_CtorInit
  - CpppAst_CtorInitList
  - CpppAst_Decl
    - CpppAst_AsmDecl
    - CpppAst_ExceptionDecl
    - CpppAst_FctDecl
    - CpppAst_LinkageDecl
    - CpppAst_ObjDecl
      - CpppAst_ArgDecl
    - CpppAst_TemplateDecl
  - CpppAst_DeclList
    - CpppAst_ArgDeclList
    - CpppAst_MembList
  - CpppAst_DeclSpec
    - CpppAst_ExceptionSpec
    - CpppAst_KeyDeclSpec
      - CpppAst_LinkageSpec
      - CpppAst_StorageSpec
  - CpppAst_DeclSpecList
  - CpppAst_Declarator
    - CpppAst_AddrDeclarator
    - CpppAst_ArrayDeclarator
      - CpppAst_NewArrayDeclarator
    - CpppAst_BitDeclarator
    - CpppAst_EmptyArrayDeclarator
    - CpppAst_FctDeclarator
    - CpppAst_InitDeclarator
    - CpppAst_MemberPtrDeclarator
    - CpppAst_PtrDeclarator
    - CpppAst_PureDeclarator
  - CpppAst_DeclaratorList
  - CpppAst_ERROR
  - CpppAst_Enumerator
  - CpppAst_Expr
    - CpppAst_BuiltinCast
    - CpppAst_Cast

CpppAst_Character
CpppAst_Delete
CpppAst_DeleteArray
CpppAst_EmptyExpr
CpppAst_ExprList
CpppAst_Float
CpppAst_IfThenExpr
CpppAst_Integer
CpppAst_Name
CpppAst_QualName
CpppAst_RootQualName
CpppAst_SimpleName
CpppAst_PrivateName
CpppAst_SpecialName
CpppAst_ConstructorName
CpppAst_ConversionName
CpppAst_DestructorName
CpppAst_OperatorName
CpppAst_TemplateClassName
CpppAst_New
CpppAst_OpExpr
CpppAst_BinaryExpr
CpppAst_SpecialExpr
CpppAst_BoolExpr
CpppAst_CommaExpr
CpppAst_IndexExpr
CpppAst_CallExpr
CpppAst_ConstructExpr
CpppAst_UnaryExpr
CpppAst_AddrExpr
CpppAst_MemPtrExpr
CpppAst_MemRefExpr
CpppAst_PtrMemPtrExpr

CpppAst_PtrMemRefExpr
CpppAst_SetIniter
CpppAst_Sizeof
CpppAst_SizeofType
CpppAst_String
CpppAst_ThrowExpr
CpppAst_FctDeclHead
CpppAst_Handler
CpppAst_HandlerList
CpppAst_Program
CpppAst_ProtectSpec
CpppAst_Statement
CpppAst_BreakStmt
CpppAst_CaseStmt
CpppAst_CmpdStmt

- CpppAst_ContinueStmt
- CpppAst_DeclStmt
- CpppAst_DefaultStmt
- CpppAst_DoStmt
- CpppAst_ExprStmt
- CpppAst_FctBody
  - CpppAst_EmptyFunction
- CpppAst_ForStmt
- CpppAst_GotoStmt
- CpppAst_IfStmt
  - CpppAst_IfElseStmt
- CpppAst_LabelStmt
- CpppAst_ReturnStmt
- CpppAst_StmtList
- CpppAst_SwitchStmt
- CpppAst_TryStmt
- CpppAst_WhileStmt

CpppAst_SwitchExpr

CpppAst_TestExpr

CpppAst_Type

- CpppAst_ClassDef
  - CpppAst_StructDef
  - CpppAst_UnionDef
- CpppAst_ClassSpec
  - CpppAst_ClassDefSpec
    - CpppAst_UnionDefSpec
  - CpppAst_UnionSpec
- CpppAst_EnumDef
- CpppAst_EnumList
- CpppAst_EnumSpec
- CpppAst_NamedType
- CpppAst_NewType
- CpppAst_PrimType

CpppAst_TypeList

CpppObjectInfo

- CpppObject_Class
- CpppObject_Enum
- CpppObject_Enumerant
- CpppObject_Exception
- CpppObject_Function
  - CpppObject_BuiltinFunction
- CpppObject_Inherited
- CpppObject_Template
  - CpppObject_ClassTemplate
  - CpppObject_FctTemplate
- CpppObject_Typedef
- CpppObject_Union
- CpppObject_Variable

CpppScopeInfo

- CpppTypeInfo
  - CpppType_Address
  - CpppType_Array
    - CpppType_FixedArray
  - CpppType_BitField
  - CpppType_Class
    - CpppType_Union
  - CpppType_Enum
  - CpppType_Function
    - CpppType_MemberFunction
  - CpppType_Pointer
    - CpppType_MemberPointer
  - CpppType_Primitive
  - CpppType_Qualified

# CTRANS USERS' GUIDE

## *Introduction*

This application converts structs in C or C++ programs into classes, and turns functions into members of these new classes.

To perform a conversion, you have to select the struct that you want to become a class, and also select all the functions that you wish to be members of the new class. Ctrans will then produce a class definition and remove the struct definition, and will update all the references to members of the new class, throughout the program.

You will find below:

- A sample ctrans session,
- A list of the kinds of functions that can be converted,
- A list of cases where conversions cannot be done,
- A guide to possible bugs in the output produced.

## *Sample session*

```
%ctrans cont_list.c
```

ctrans is invoked from the Unix prompt with the command 'ctrans', and an optional filename argument.

```
Ctrans C to C++ conversion program.
type ? for list of commands.
Line 21 of file "cont_list.c": Name `malloc' is undefined
Line 43 of file "cont_list.c": Name `error' is undefined
```

Any errors in the input program will be displayed.

```
Ctrans:?
Possible commands are:
  (c)onvert a struct into a class, with member funcs.
  (d)ump the abstract syntax tree of the program.
  (p)rint the program tree as code.
  (r)ead [filename] - read in a new C/C++ file.
  (s)ave [filename] - save the tree as code.
  ? - display this list.
  (h)elp - display a brief help message.
  (q)uit the program.
```

Enter '?' to see this list of possible commands. If a file had not been used as an argument, the 'r' command can be used to read one in However, only one file can be used in each session. The 'd' command is not likely to be useful - it displays the internal representation ctrans

uses for the program. The most important command is ‘c’, to start a conversion.

```
ctrans:c
Choose the struct to become a class:
Enter choice number from the following
list, or m to return to the main menu.
[1]  ListNode
[2]  ListTag
```

First, you must choose the struct which is to become a class, by entering it's number from the list displayed. Ctrans uses a modified prompt if it’s not in the main menu mode: here ‘-c’ is added to the prompt, to indicate a ‘c’ command is being performed.

```
ctrans-c:1
Class to be set to: ListNode
Choose member functions, one at a time,
by:  - entering a number from the list,
     - entering m for the main menu,
     - entering t complete selection, and
       start the transformation
[1]  isEmpty
[2]  MakeNode
[3]  isEmpty
[4]  length
[5]  head
[6]  add
```

Next, a list of all the functions defined in the input program is displayed. You should enter the number of the first function that you want to become a member.

```
ctrans-c:2
Function to become member:MakeNode

Choose member functions, one at a time,
by:  - entering a number from the list,
     - entering m for the main menu,
     - entering t complete selection, and
       start the transformation
[1]  isEmpty
[2]  MakeNode
[3]  isEmpty
[4]  length
[5]  head
[6]  add
ctrans-c:t
```

Here, only one function is to become a member. The ‘t’ command is used to start the transformation.

```
Line 43 of file "cont_list.c": Name `error' is undefined
Line 52 of file "cont_list.c": Name `operator new' is undefined
Transformation successful
ctrans:p
```

‘p’ prints the new program to the screen.

```
typedef  int  ItemType;

int  isEmpty(char  *s) {
   return s[0] == 0;
}

class ListNode {
   private:
      ItemType item;
      struct ListNode *next;
   public:
```

```
      ListNode(ItemType it);
} ;

struct ListTag {
      struct ListNode *first;
} ;

typedef  struct ListTag List;

ListNode::ListNode(ItemType it) {
   item = it;
   next = 0;
}

int  isEmpty(List l) {
   return l.first == 0;
}

int  length(List l) {
   int  i = 0;
   while (isEmpty(l))  {
      l.first = l.first->next;
      ++i;
   }
   return i;
}

ItemType head(List &l) {
   if (isEmpty(l))
      error("attempt to remove head of empty list");
   else  {
      ListNode *ln = l.first;
      l.first = l.first->next;
      return ln->item;
   }
}

void  add(List &l, ItemType it) {
   ListNode *ln =  new ListNode it;
   ListNode *temp;
   temp = l.first;
   while (temp->next != 0)
      temp = temp->next;
   temp->next = ln;
}

ctrans:c
```

Now, another conversion can be performed, if there are any structs left in the program.

```
Choose the struct to become a class:
Enter choice number from the following
list, or m to return to the main menu.
[1]  ListTag
ctrans-c:1
Class to be set to: ListTag
Choose member functions, one at a time,
by:  - entering a number from the list,
     - entering m for the main menu,
     - entering t complete selection, and
       start the transformation
[1]  isEmpty
[2]  constructor class ListNode
[3]  isEmpty
[4]  length
[5]  head
[6]  add
ctrans-c:3
```

Note that the ‘isEmpty ‘function in the program is  overloaded. The order that they appear on this list is the same as in the

program, so function 3 is the ‘isEmpty’ which takes a ‘List’ argument.

```
Function to become member:isEmpty

Choose member functions, one at a time,
by:  - entering a number from the list,
     - entering m for the main menu,
     - entering t complete selection, and
       start the transformation
[1]  isEmpty
[2]  constructor class ListNode
[3]  isEmpty
[4]  length
[5]  head
[6]  add
ctrans-c:4
Function to become member:length

Choose member functions, one at a time,
by:  - entering a number from the list,
     - entering m for the main menu,
     - entering t complete selection, and
       start the transformation
[1]  isEmpty
[2]  constructor class ListNode
[3]  isEmpty
[4]  length
[5]  head
[6]  add
ctrans-c:5
Function to become member:head

Choose member functions, one at a time,
by:  - entering a number from the list,
     - entering m for the main menu,
     - entering t complete selection, and
       start the transformation
[1]  isEmpty
[2]  constructor class ListNode
[3]  isEmpty
[4]  length
[5]  head
[6]  add
ctrans-c:6
Function to become member:add

Choose member functions, one at a time,
by:  - entering a number from the list,
     - entering m for the main menu,
     - entering t complete selection, and
       start the transformation
[1]  isEmpty
[2]  constructor class ListNode
[3]  isEmpty
[4]  length
[5]  head
[6]  add
ctrans-c:t
Function call with 'this' as an argument found
in MFTB where the CTB argument is by value.
This call may modify the CTB argument -
*Failed* to transform this function.
```

Note here that a function has not been transformed. The output below shows that this is the ‘length’ function. This is because it calls ‘isEmpty’, and ‘isEmpty’ may modify it argument. If it did, this would cause errors in

the converted program if ‘length’ were to become a member function.

```
Line 34 of file "cont_list.c": Illegal expression: ambiguous call to
function `isEmpty'
Line 43 of file "cont_list.c": Name `error' is undefined
Transformation successful
ctrans:p
typedef  int  ItemType;

int  isEmpty(char  *s) {
   return s[0] == 0;
}

class ListNode {
   private:
      ItemType item;
      struct ListNode *next;
   public:
      ListNode(ItemType it);
} ;

class List {
   private:
      struct ListNode *first;
   public:
      int  isEmpty()const  ;
      int  length()const  ;
      ItemType head();
      void  add(ItemType it);
} ;

ListNode::ListNode(ItemType it) {
   item = it;
   next = 0;
}

int  List::isEmpty()const   {
   return first == 0;
}

int  length(List l) {
   int  i = 0;
   while (isEmpty(l))  {
      l.first = l.first->next;
      ++i;
   }
   return i;
}

ItemType List::head() {
   if (isEmpty())
      error("attempt to remove head of empty list");
   else  {
      ListNode *ln = first;
      first = first->next;
      return ln->item;
   }
}

void  List::add(ItemType it) {
   ListNode *ln =  new ListNode it;
   ListNode *temp;
   temp = first;
   while (temp->next != 0)
      temp = temp->next;
   temp->next = ln;
}

ctrans:s oo_list.cc
```

The ‘s’ command is used to save the converted program as ‘oo_list.cc’.

```
ctrans:q
%
```

### *Conversions performed*

A list of the kinds of structs and functions that can be converted is given below. The following terms are used in this list:

CTB - The class-to-be, i.e. the struct that is to become a class
MFTB - A member-function-to-be

- struct definitions
- struct definitions combined with a ‘typedef’
- MFTBs with a ‘&’ or ‘*’ CTB argument
- MFTBs with a value CTB argument. In this case, the function will only be converted if it does not appear to change the CTB argument. This is because such changes would affect the actual parameter used if it became a member function.
- MFTBs returning a CTB variable by value. These will be converted into constructor functions, and so the return statement and the local variable that is returned will be removed from the program.
- MFTBs returning a pointer to a CTB variable. These will also be converted into a constructor, with the return statement and local variable removed. Ctrans will additionally attempt to find and remove any memory allocation performed for the pointer returned.

### *Conversions not performed*

- MFTBs with more than one CTB argument
- MFTBs that return a CTB, and also have a CTB argument
- MFTBs returning a ‘&’ CTB variable
- struct definitions combined with ‘typedef ..... * *typedef-name’*
- MFTBs that return a CTB, but have more than one return statement
- MFTBs without any CTB argument
- MFTB function prototypes

### *Possible bugs in output*

- If an MFTB is not converted - for any of the reasons given in the two lists above - a function prototype for it will still be created in the definition of the new class.
- If an MFTB returns a CTB, and uses a local variable for this, but this variable is declared as part of a list of CTB variables, then its declaration will not be removed.

- All variables in the new class are declared as “private:”. Therefore there will be a compilation error if any of these variables are accessed from a function that is not a member.
- It is possible that some member references within functions will not be updated.
- ‘for’ loops are printed with an extra (erroneous) semicolon after the *continue* statement in the expression.
- Expressions are not bracketed, so they may not be evaluated correctly.
- Output will be very unreliable for the following language constructs: casts, as declarations, initialisations of set variables, base specifiers for classes, ‘new’ memory allocation statements, all statements for exception handling, all statements concerning templates.

# PATTERN WRITERS' GUIDE
# and pt users' guide

## *Pattern writing*

### Overview

Patterns are used for specifying how to transform a program. These specifications are used as input for the application ctrans, which converts C code into C++ code. The patterns are therefore targeted for transforming C into C++, but with modifications, it should be possible to perform other conversions.

The idea of pattern matching is used, where the transformations are specified by two patterns of syntax tree nodes: an if-find pattern and a replace-with pattern. The if-find pattern is compared with the syntax tree of the program to be transformed, and if any section matches the if-find pattern, then this section is replaced with the replace-with pattern.

The requirements for a tree section to match a pattern are covered in detail later, but essentially the root nodes of the pattern and the tree have to be of the same 'type', and every child node of the tree section must match the equivalent child of the pattern. The type of a node is the kind of (C++) syntax structure that it represents, for example a function definition, or a list of declarators.

Patterns are in fact more complicated; the structure that specifies a transformation is a trans-spec. This consists of a list of if-replace-specs, and each of these consists of an if-find pattern and a replace-with pattern. This is to enable more than one transformation to be attempted for the same tree, for example to convert both data member accesses and member function calls within the body of a function. For each node in the syntax tree, if the if-find pattern belonging to the first if-replace-spec does not match it, then the next if-replace-spec will be tested, and so on.

The individual if-find and replace-with nodes can have several different properties in addition or instead of their type. The components of an if-find node specify how syntax tree nodes can match it, other than just having the same name. Similarly, components of a replace-with node specify the kind of syntax tree node to create in the transformed tree. The possible combinations of components for each node are given in 'Match and replace specifications' below, but first the meaning of each of the components will be described.

### If-find node components

- name

A text name, either "*ANY*" or the required type for the syntax tree node

- star-number

  The presence of this means that any node in the syntax tree, with any sons, will match this if-find. Also the node can be referred to in other patterns by this number.

- special match

  This is a mechanism for specifying additional requirements for a syntax tree node to mach. In ctrans, the special-match will in fact be used as an argument to the sMatch() function, which has access to certain information about the desired transformations.

  A list of the special matches currently available in ctrans is given below, using the following terms:

  CTB - The class-to-be, i.e. the struct that is to become a class
  MFTB - A member-function-to-be

| Special match | Requirements for syntax tree node |
|---|---|
| MFTB | identifier for an MFTB |
| DirCTB | type name for CTB directly |
| PtrCTB | type name for pointer to CTB |
| AnyCTB | type name for CTB either |
| CTBtag | tag for a CTB struct |
| VarDirCTB | variable, type = CTB |
| VarPtrCTB | variable, type = pointer to CTB |
| VarAnyCTB | variable, type = (either) CTB |
| VarDitCTBEq(*) | variable, type = CTB, name = *node name |
| VarPtrCTBEq(*) | variable, type = pointer CTB, name = *node name |
| EqualTo(*) | variable, name = *node name |
| MemPub | data member of CTB, protection = public |
| MemPri | data member of CTB, protection = private |
| MemProt | data member of CTB, protection = protected |
| AssignError | <causes error message to be printed> |
| CallError | <causes error message to be printed > |

- sons

  A list of if-find nodes, each of which must have an equivalent in the syntax tree node being tested, and all equivalent sons must also match

- trans-spec

  Usually used with a star number, is an embedded trans-spec to apply to the node and it's sons. If the trans-spec modifies this node or it's sons, then the modified version will be used whenever the star number is referenced.
  The trans-spec can be modifying (M) or conditional (C) or both. If it is conditional, this means that the if-replace-specs it owns may cause the trans-

spec to not match the node (see below). Note that it is the if-find node that must have either one or both of the tags (M and C) if it has a trans-spec.

**Replace-with node components**

- name
  The name the created node is to have.

- star-number
  Identifies a node from the original tree.

- spec-create
  A special create macro similar to the special match, that uses an sCreate() function in ctrans to create the desired node.

| Special create | Resultant syntax tree node |
|---|---|
| CTBName | identifier for the name of the new class |
| MFTBHdrs | list of function prototypes for all the MFTBs, suitable for adding to the definition of the new class |

- sons
  A list of replace-with nodes. The nodes created for each are added as sons of the node created for this replace-with

- REMOVE_SON
  Specifies that this node in the parent of the equivalent tree node is to be removed.

- CONCAT
  Specifies that the sons of this node are to be added in place of this node in the equivalent tree node.

**Trans-spec components**

The only component of a trans-spec is the list of if-replace-specs.

**If-replace-spec components**

The if-replace-spec has two components in addition to the if-find node and replace-with node, the first of which is a number-to-find. This will normally be -1, indicating that any number of patterns matching the if-find node can be found in the syntax tree. For a modifying only trans-spec (M), all it's if-replace-specs must have -1 as their number-to-find.

However, if the number to find is zero or greater, then exactly that number of sections in the syntax tree matching the if-find node must be found. If a different number is found, then the parent trans-spec will fail to apply. If any of these kind of if-replace-specs are used in a trans-spec, then it must have a conditional (C) tag. Note that if the number-to-find is zero for every if-replace-spec in a trans-spec, then it is conditional only as occurrences of any of the if-replace-specs will mean that the trans-spec is not applied.

| Number to find | Meaning |
|---|---|
| -1 | any number is OK |
| -2xx | at least xx must be found |
| -3xx | up to xx can be found |

The second component is a list of star-numbers to remove. This is to allow the same star numbers to be used in different if-replace-specs, and so normally this list will be of all the star numbers that the pattern uses. However, if a star node has to be accessed from a different if-replace-spec in the trans-spec, then this node's number should be left off the list and put onto the remove list of the next if-replace-spec up.

**Match and replace specifications**

| If-find | Requirements to match |
|---|---|
| name + star-num + [(M)] | names same |
| name + [sons] | names same + all sons match |
| name + (C) + [ star-num \| (star-num + (M)) ] | names same + trans-spec matches |
| name + special-match + [star-num] | names same + special-match matches |

| Replace-With | Node created |
|---|---|
| star-num | star-node (possibly modified) |
| star-num + sons | star-node with sons added after existing sons |
| special-create | special-create node |
| REMOVE_SON | remove this son from it's parent |
| CONCAT + sons | add sons in place of this son in it's parent |
| name | node with same type-name |
| name + sons | node with same name, sons = create(sons) |

**Notes**

The order that if-replace-specs are listed in is important - the first one that matches a given section of tree will be applied. Later if-replace-specs in the list will not be checked against a node if an earlier if-replace-spec has already matched it.

Star numbers will usually be visible in the if-replace-spec where they are found, and in all child nodes and trans-specs. However, if a star number is left off the list of stars-to-remove, it will be visible throughout all of the trans-spec in which owns the if-replace-spec.

In ctrans, a ‘top-level’ trans-spec is declared, which is what is actually matched against the program. This trans-spec must always apply, and so any if-replace-specs added to it MUST have ‘-1’ as their number-to-find.

Note that a trans-spec attached to an if-find node will work even if the equivalent syntax tree node actually matches the first if-replace-spec in the trans-spec.

The syntax used for all node types is that of the Cppp parser/analyser. Familiarity with this syntax is a prerequisite for writing patterns for ctrans, and it is recommended that many examples of the syntax trees produced by Cppp are examined. Cppp itself is available at ftp://wilma.cs.brown.edu:/pub/cppp.tar.Z, and but the ctrans command ‘d’ will also display the syntax tree of the current program.

### *Pt*

The pt (pattern translator) tool takes an input file that describes a pattern and produces a C++ file containing the implementation of this pattern. The input file should be in the format described below in ‘ctp syntax’, and the output file should be linked in to ctrans (see ‘Adding patterns to ctrans’ below).

pt takes two command line arguments, the name of the input file and the starting number:

pt *filename start-num*

If the file is called *filename*.ctp, the output file will be called *filename*.cc, and note that pt expects all filenames to have a three-character ‘.’ suffix.

The start number is the numerical part of the identifier given to the top level pattern object produced (which must be a trans-spec or an if-replace-spec). This allows name of this object known so that it can be included in other parts of the program.

### *Ctp syntax*

The syntax of pt input files is quite similar to the text output the Cppp produces. Each node appears on a separate line, and the nodes appear in the order of a depth-first, pre-order traversal of the tree.

The sons of a node appear immediately below it, and with one extra step of indentation. Trans-specs are treated as a special case of a son, and appear as a line with the text

"#TransSpec"

on it. Each if-replace-spec is then a son of the trans-spec node, and these are represented by

"#IfReplaceSpec num-to-find [stars-to-remove]"

The num-to-find is an integer and the optional stars to remove list is a list of integers, separated by commas (but not spaces).

Anything without a '#' is assumed to be a node, and no distinction is necessary between if-find and replace-with nodes. Each component of the node has a special format, and these can appear on the line for the node in any order. The component formats are described below.

- Name: as text
- Second name: (e.g. typedef for KeyDeclSpec) as text, must appear after first name
- Star number: an integer preceded by an asterisk.
- Special match/create: enclosed by two underscore characters. A star-number argument is enclosed in parenthesis, e.g.
  _EqualTo(34)_
- Trans-spec: presence indicated by one or both tags, (C) and (M)
- Remove-son: name as *REMOVE_SON*
- Concat: name as *CONCAT*
- Sons: any lines below a nodes line, and indented further, are assumed to be sons

Note that:

- there must not be any spaces between the numbers in a stars-to-remove list;
- the spelling of the names of nodes must be correct - there are no checks made on it;
- the indentation must be used completely consistently, to allow pt to work out the sons of each node correctly
- NULL is allowed as the replace-with node if an if-replace-spec has 0 as the number-to-find
- and finally, C++ format is used for comment ('//', comments can appear anywhere and go to end of line).

A complete example of a ctp file is shown below. This file describes the following conversions:

typedef struct [*ctb_tag*] {
    *members*
} *ctb_type*
⇒
    class *CTBName* {
        *members*
        *MFTB-headers*

}

*members*:

*type-specifier members_all_to_be_public*

⇒

public: *type-specifier members_all_to_be_public*

*type-specifier members*

⇒

public: *type-specifier members*

```
#IfReplaceSpec  -1   1,2,3    // to be numbered 500
   ObjDecl                 // ie call 'pt typedef_struct.ctp  500'
      DeclSpecList
         KeyDeclSpec  typedef
         ClassBody
            StructDef
               ClassDefSpec
                  _CTBtag_
               *2
               MembList  (M)  *3
                  #TransSpec        //  For adding "public:" etc
                     #IfReplaceSpec -1  4,5
                                     // For public declarators
                        ObjDecl
                           DeclSpecList  (M)  *4
                              #TransSpec
                                 #IfReplaceSpec -1
                                    ClassSpec
                                       _DirCTB_
                                    _CTBName_
                           DeclaratorList (C)  (M)  *5
                              #TransSpec
                                 #IfReplaceSpec -201  6
                                    _MemPub_  *6
                                    *6
                                 #IfReplaceSpec   0
                                    _MemProt_
                                    NULL
                                 #IfReplaceSpec   0
                                    _MemPri_
                                    NULL
                        *CONCAT*
                           ProtectSpec  public
                           ObjDecl
                              *4
                              *5
                     #IfReplaceSpec -1  4,5
// For mixed declarators
// lowest common protect spec => public
                        ObjDecl
                           DeclSpecList  (M)  *4
                              #TransSpec
                                 #IfReplaceSpec -1
                                    ClassSpec
                                       _DirCTB_
                                    _CTBName_
                           DeclaratorList  *5
                        *CONCAT*
```

```
                        ProtectSpec  public
                        ObjDecl
                           *4
                           *5
            ClassFcts  *1
      DeclaratorList
         InitDeclarator
            SimpleName
            ExprList
   ObjDecl
      DeclSpecList
         ClassBody
            ClassDef
               ClassDefSpec
                  _CTBName_
               *2
               *3
                  _MFTBHdrs_
                    // will add MFTB hdrs as new sons of *3
            *1  // ClassFcts
      DeclaratorList
```

### *Adding patterns to ctrans*

Adding new special macros will first be covered briefly, and then how to link a pt output file with ctrans will be discussed.

When writing new patterns, it is often useful to create a new special match or special create. This will require some knowledge about the structure of ctrans and the nature of Cppp nodes.

Firstly, the enumerator in ct_local.h needs to have the new case added to it. The enumerator CtMatch specifies all possible special matches, and CtCreate specifies all possible special creates.

The functions isMatch and isCreate in ptmain.cc will also have to be updated to allow the new enumerants, and finally the actual application of the macros occurs in ctmanager.cc, in the functions sMatch() and sCreate().

To add a new .cc pattern file to ctrans, a number of files need to be updated.

If the pattern is for an MFTB, the header part of the function needs to be included in the pattern file func_hdrs.ctp, which is used for creating the function prototypes in to new class definition.

The Makefile must be updated to include the new .cc file.

The root if-replace-spec of the new pattern must be declared in the ct_patterns.h file, where its name will be “Irs_<pt-start-num-used>“.

Finally, this if-replace-spec must be added to the trans-spec declared in ctpattern.cc. It must be added to the array of if-replace-specs, and also the number of if-replace-specs in the CtTransSpec constructor must be changed appropriately.