Uppsala Master's Thesis in
Computing Science 223
Examensarbete DV3
2002-08-30
ISSN : 1100-1836

Implementation of the ESRA Constraint Modelling Language

Simon Wrang

Information Technology Department/Computing Science
Uppsala University, Box 337, S-751 05 Uppsala, Sweden

This work has been carried out at the
Department of Information Science
Uppsala University, Box 513, 751 20 Uppsala, Sweden

Abstract

This report describes my Master's thesis project of implementing a constraint modelling language called ESRA. ESRA adds new features to an already existing constraint modelling language called OPL. With the help of the tools JLex and JavaCUP, I have created a generic compiler that easily lets one change the grammar and the ESRA-to-OPL rewrite rules of the ESRA language. I show how to use JLex and JavaCUP in order to combine them, to make a two-pass compiler with them, and to detect errors with them. I also show how I created a rule file system that translates rules in a natural format into Java code. In a second phase of the project, with the help of a tree algorithm, I made the compiler non-deterministic, i.e., the compiler can take one input program and translate it into *several* output programs. The generated programs accomplish the same task but are implemented in different ways; they therefore have different execution times, letting the user or the compiler select the fastest one.

Supervisor: Brahim Hnich and Pierre Flener
Examiner: Pierre Flener
Passed:

Contents

Chapter 1

Introduction

1.1 Background

At the Department of Information Technology and the Department of Information Science at Uppsala University in Sweden, a research group called ASTRA is working on designing a new programming language called ESRA. ESRA is a constraint modelling language and is an extension of an already existing constraint modelling language called OPL, see [5]. ESRA keeps the good parts of OPL and adds some new features to make it more efficient to use. So far the ESRA grammar and parts of the ESRA-to-OPL translation rules have been determined, see [6], section 4.2. What remains is to actually implement the language — to create the compiler.

1.2 About This Project

The main goal of my project is to implement the ESRA language. This takes place in two phases. In the first phase I create a compiler and complete the translation rules. The compiler is able to translate from ESRA into OPL. In the second phase I make the compiler non-deterministic. This means that it is able to generate several output programs from one input program. Also, an additional goal is to make the compiler as flexible as possible — it should be easy to update the grammar and the translation rules.

1.3 This Report

This report is divided into five chapters. The first chapter is the current one and serves as an introduction to my work. The second chapter goes through the basics needed to understand this project. A reader not familiar with compilers or constraint programming should read this chapter. The third and the fourth

chapter deal with the problem and solution of the first and the second phase respectively. The fifth and final chapter is the conclusion of the project.

At the end of the report there are two manuals, listed as appendix A and appendix B, which pertain to the second version of the ESRA application, i.e., that of phase 2. The first manual is for *users* of the ESRA application, while the second manual is for *programmers* who want to modify the code of the ESRA application.

Also, a diskette containing the ESRA application in question is available together with the report.

1.4 Acknowledgments

I thank the people who have helped me with this project. Most of all, I thank Brahim Hnich, who has served as my day-to-day supervisor during this project. He has always been available to help me and has offered a lot of good advice. I also thank my formal supervisor, Pierre Flener, who has helped me with writing the report and guided my project.

Chapter 2

Basics

2.1 Constraint Programming

The ESRA language, which I am implementing, is a constraint programming language. In this section I explain what constraint programming is for those readers who are not familiar with the term.

A constraint program is structured in regard to modelling a problem. It is unlike a program in an imperative language, like C, Pascal and Java, in which one models the procedure of finding the solution. A constraint program lets the user concentrate on defining the problem, and then does all the work of solving the problem itself.

In *constraint programs*, in a simplified view, one has a set of variables, *domains* for the variables, and a set of *constraints*. The domains are finite sets of values that define the types of the variables. The constraints are logical expressions that contain the variables and restrict their possible values. The underlying core, called the *constraint solver*, finds the values for the variables so that all the constraints are fulfilled. It is rather like solving an equation system.

To understand this better, here is an example of what a constraint program in the OPL language can look like:

```
var int x in 1..5;
var int y in 2..3;
solve {
  x>y;
  x+y=7;
}
```

This program consists of six lines. The first two lines declare two variables: `x`, an integer between 1 and 5; and `y`, an integer between 2 and 3. The fourth and fifth lines are the constraints.

The constraint solver now solves this program. When run it prints the following result:

```
solution 1: {x=4;y=3}
solution 2: {x=5;y=2}
```

This was just a simple example using simple variables and simple constraints. One can also use sets, arrays, records, for-statements and other structures typically found in an imperative language. This lets one model more complex problems. Read about this in section 2.4 about ESRA and OPL.

Those readers familiar with logic programming, such as Prolog, might wonder what the difference is from constraint programming. The answer is that constraint programming is the next evolving step after logic programming. Constraint programs let one do more things that logic programs cannot.

2.2 Compiling - Implementing a Language

My task is to implement the ESRA language, to write its compiler. In this section I explain what a compiler is and describe the normal procedure to write one.

What is a compiler To implement a new language means in some sense to make it understandable to the computer. The way to do this is to have something that can translate from the new language into a language that the computer already understands. This something is called the compiler.

In my case I am making a compiler that translates into the OPL-language. The OPL-language in itself cannot be understood by the computer, but it in turn has a compiler that translates into a language that can be understood.

How to write a compiler To write a compiler one first needs to be clear on what the grammar of one's language is. The *grammar* is a set of rules that together state what a program in the language must look like. For example, let's say one has a language that lets one write simple arithmetic expressions with addition and subtraction of numbers and identifiers. Possible programs in this language would be: `4-acc+10`, `a+b+c-3` and just `12`. Here is what the grammar would look like:

```
EXPR -> EXPR PLUS EXPR
      | EXPR MINUS EXPR
      | ARGUMENT
ARGUMENT -> ID
          | NUMBER
```

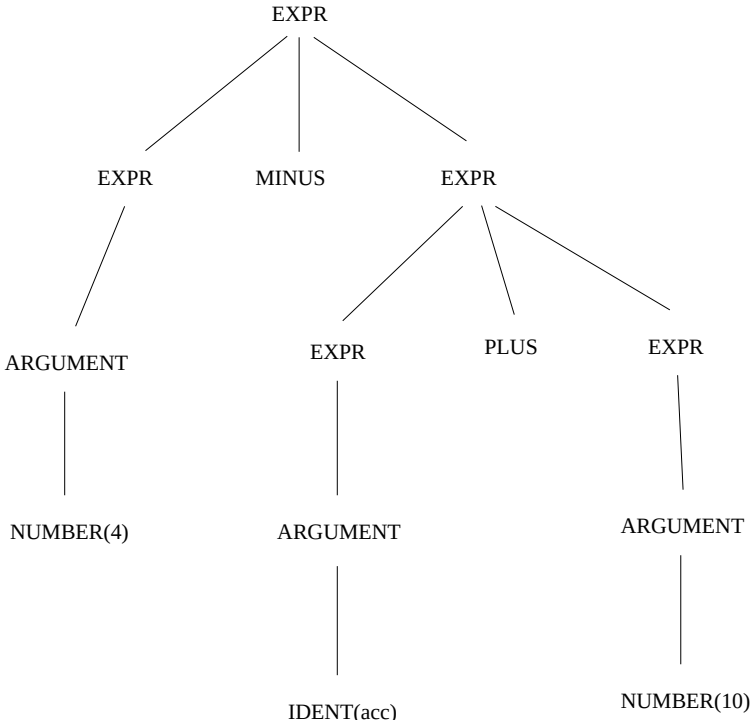
Each paragraph with an arrow is called a *rule*. To the left of the arrow is the term that produces something; it is called a *non-terminal*. To the right of the arrow are the terms that are being produced. Terms that occur here that don't produce anything are called *terminals*. The vertical bar separates different alternatives to what is produced; these alternatives are called *productions*.

Once the grammar has been determined it is possible to start writing the actual compiler. The compilation process usually is divided into three steps: *tokenizing*, *parsing* and *translation*.

Tokenizing The first step in the compilation process is to tokenize the input program. Tokenizing means to divide the stream of characters into larger chunks, *tokens*, where every token can consist of one or more characters. A good set of tokens for the expression language would be the plus sign, the minus sign, identifiers and numbers. The program `4-acc+10` would generate the following tokens:[1]

```
NUMBER(4), MINUS, IDENT(acc), PLUS, NUMBER(10)
```

Parsing Once the tokenizing is done, the next step is to parse the tokens. Parsing means putting the tokens in a tree structure called the *parse tree*, which reflects the grammar of the language. The leaves contain the tokens of the input program; the nodes contain the non-terminals of the productions being used to generate the input program. In the case of the input program `4-acc+10`, here is what the parse tree would look like:



[1] Note that tokens can have values, as is the case with numbers and identifiers.

Translation Once the parse tree has been constructed it is normally easy to do the translation. One simply traverses the parse tree and constructs the translated result as one goes along.

Two Passes Normally when one is constructing a compiler, one talks about how many passes through the code one has to make. In the above example one pass would be enough. However, for most programming languages more than one pass is needed. A programming language that lets the user declare variables typically requires two passes. In the first pass the declarations are processed and data about them are stored. The thing that stores the data is usually denoted as a *symbol table*. In the second pass the compiler uses the data in the symbol table to do the actual translation.

Help Tools Writing a compiler on one's own can be a very tedious job. That is why there exist help tools to ease one's work. The most common ones are *Lex* and *Yacc* for creating compilers implemented in the C-language. They let the user write a specification file in a certain format for the tokenization and the parsing respectively, which then become converted into actual runnable C programs. In my project I have used the tools *JLex* and *JavaCUP*, which are similar tools, but for creating compilers implemented in the Java language.

2.3 Help Tools JLex and JavaCUP

For writing the compiler I use two help tools: JLex for the tokenizing and JavaCUP for the parsing and the translation.

JLex JLex expects a specification file from which it will generate a runnable Java class named Ylex. The specification should be in a special format and include information on how the different tokens should be created. Here is what the specification file for the example in the previous section can look like:

```
import java.lang.System;
%%
DIGIT=[0-9]
NUMBER=({DIGIT})*
WHITE_SPACE=([\ \n\r\t\f])+
%{
  public void printToken(String t) {
    System.out.println("Token: "+t);
  }
%}
%%
<YYINITIAL> "+" {
  printToken("PLUS");
}
```

```
<YYINITIAL> "-" {
  printToken("MINUS");
}
<YYINITIAL> {NUMBER} {
  printToken("NUMBER,"+yytext());
}
<YYINITIAL> {IDENT} {
  printToken("IDENT,"+yytext());
}
<YYINITIAL> {WHITE_SPACE} {
}
```

I now briefly describe the structure of the specification file. For more details read the JLex manual [1].

The specification file is divided into three parts separated by %%. Whatever one puts in the first part is inserted at the top of the generated class. This is a good place to put imports and package declarations.

In the second part one can create *macro declarations* and insert one's own Java code. Macro declarations tie names to regular expressions that can be used lower down in the third part. The Java code is any code one wants to have inserted into the body of the generated class.

In the third and final part one specifies how the different tokens are defined. A token's definition can be a string, a macro declaration or a regular expression. When scanning through the input characters one of the token definitions sooner or later matches. The Java code inside that definition is then executed.

The above specification file, when run through JLex, generates a Java program that prints out all the tokens in a list. For the input `4-acc+10` the following is printed:

```
Token: NUMBER,4
Token: MINUS
Token: IDENT,acc
Token: PLUS
Token: NUMBER,10
```

JavaCUP JavaCUP is the help tool for generating the parser. Like JLex it lets the user write a specification file. With a certain format one specifies the grammar of the language. With every production in the grammar one also writes code, called the *semantic code*. It is executed for those productions that are used to produce the input program. It is in the semantic code that the actual translation gets done.

Here is what the JavaCUP specification file can look like for the example language in the previous section:

```
import java_cup.runtime.*;
```

```
terminal SEMI, PLUS;
terminal String NUMBER, IDENT;
non terminal String EXPR, ARGUMENT;
EXPR ::= EXPR:e1 PLUS EXPR:e2
           {: RESULT = e1+"-"+e2; :}
       | EXPR MINUS EXPR
           {: RESULT = e1+"+"+e2; :}
       | ARGUMENT:a
           {: RESULT = a; :}
       | ARGUMENT ::= ID:id
           {: RESULT =
                id.toUpperCase(); :}
       | NUMBER:n
           {: RESULT = n; :}
```

First come imports and package declarations, which like JLex are inserted at the top of the generated parser class. Then follow declarations of the terminals and non-terminals used in the grammar. Finally there is the declaration of the grammar.

To understand how the grammar is specified one first needs to understand how the parsing works. The parsing process is recursive. It starts at the leaves and works itself up to the root of the parse tree. The result from parsing one production is passed up to its parent production in the next level of the tree. When having reached the root node of the tree, the translation is complete and the value in the root node is the translation result.

In the grammar, the semantic code is specified inside `{:` and `:}` for every production. The result of a production should be assigned to the `RESULT` variable which is predefined. If one looks at the grammar one also sees names next to the terms separated by a colon. These are called *labels* and identify the results from parsing those terms.[2] These results can then be used in the semantic code by referring to the labels.

What the above example parser does is to switch plus signs and minus signs with each other and also capitalize all identifiers. For the input `4-acc+10` the translated result would be `4+ACC-10`.

Finally, one can also insert one's own Java code into the parser classes that is generated. There are actually two commands for this: `parser code {: :}` and `action code {: :}`. The reason for this is that JavaCUP generates several files. Among them is one that holds the main engine of the parser, named parser.java, and another that holds the semantic code of the parser, named CUP$parser$action.class.

For more details on how to use JavaCUP, read the JavaCUP manual [2].

[2] Also unparsable terminals - tokens - can have labels. In this case the label refers to the internal value of that token.

2.4 ESRA and OPL

In my compiler ESRA is translated into OPL. In this section I give a more detailed description of these two languages. I start by describing OPL and then give a description of what ESRA adds to OPL. I also discuss the rules used to translate ESRA into OPL. At the end, I describe a simplified version of the ESRA language and a language called OPL+, which are used later in the second phase of the compiler.

OPL OPL is a constraint programming language. We have already seen an example of a simple OPL program in section 2.1 about constraint programming. Like that and other OPL programs, they have a basic structure which consists of five main parts:[3] data (and type-) declarations, variable declarations, an objective, constraints and display statements.

Data declarations are the hard coded data that are used in one's program. For declaring data, most of the basic data types that one finds in procedural languages also exist in OPL; there are integers, floats, strings, arrays, and so on. OPL also has two special types that are used frequently: sets and ranges. Sets are what they sound like; ranges are defined by two numbers, a and b, where b is bigger than a. Closely related to data declarations are type declarations. Among the new types that OPL lets one declare are records (structs) and enumerations. Here are some examples:

```
a) int n = 8;
b) float f = 3.2;
c) struct Point {
     int x;
     int y;
   };
d) enum Days = {Mo,Tu,We};
e) int A[1..4] = [1,3,5,7];
f) range r 1..10;
g) {int} s1 = {1,2,3};
```

By putting the `var` keyword in front of a data declaration one turns it into a variable declaration. The variables are what OPL tries to find solutions for. Most of the data types can be variables, even arrays. However, sets and ranges in OPL cannot. Here are some variable declarations:

```
a) var int count in 1..10;
b) var int grades[1..27,1..8] in 1..10;
```

The third part of an OPL program is the *objective*. It is in the objective that one states one's goal. There are three basic types of objectives: *solve*, *minimize*

[3] There are actually other parts as well but they are of less importance, at least for this project.

and *maximize*. With solve one simply states that one wants to find all the solutions that satisfy the constraints. With minimize and maximize one also has an objective expression. With these the goal is to find the *ONE* solution that minimizes (or maximizes) the objective expression.

The constraints are the logical expressions that need to be satisfied for a valid solution. OPL offers a lot of constraints to choose from. Probably the ones most used are the relations that also exist in procedural languages: bigger than (>), less than (<), equals (=), not equal (<>), implication (=>), and equivalency (<=>). OPL also has versions of the typical procedural statements *forall* and *if-then-else*. Forall lets one specify a generic constraint that should hold for several values. The if-then-else constraint is satisfied if the if-part is true and the then-part is true, or the if-part is false and the else-part is true. Have a look at these examples:

```
x > y;
i = 1 => j = 2;
forall (i in 1..10)
  A[i] = B[i] + 1;
if x > y then x > z else y > z;
```

The final part of an OPL-program, which occurs at the bottom of it, are the display statements. When OPL finds solutions for one's variables it displays them in a default way. If one has variables that are declared with one's own defined data types, one might want to have them displayed in a different way. With display statements one can achieve this. For example, consider the declaration of the point struct given above. Using the display statement

```
display(p in Point: p.x>=0 & p.y>=0) <p.x,p.y>
```

one can have ones' points displayed as tuples. The above display statement also has the added feature that it won't display point variables that contain negative values.

Having explained all the parts that an OPL program consists of, I now show an example. The following OPL program is a model of the *Graph Coloring problem*, *GCP*. The GCP is a classical problem used in constraint programming, where the objective is to use a minimal number of colors to color every country in a map, such that no bordering countries have the same colors.

```
enum Country ...;
enum Color ...;
struct border {
  Country c1;
  Country c2;
};
{border} Borders = ...;
var int UsedColors[Color] in 0..1;
var Color Coloring[Country];
```

```
minimize
  sum(I in Color) UsedColors[I]
subject to {
  forall(B in Borders)
    Coloring[B.c2] <> Coloring[B.c1];
  forall(I in Country)
    UsedColors[Coloring[I]]=1;
};
display(I in Color: UsedColors[I]=1) <I>;
```

Looking at this program, the data declarations are the input. The input consists of two enumerations: a list of the existing countries and a list of the existing colors; and a set of existing borders. Borders are records with two fields that represent the two countries that are bordering each other. Three dots (...) in the data declarations indicate that the data are in another file.

Similarly, variable declarations is the output. In this program there are two array variables. The main solution is given to the array `Coloring` in which every country is assigned a color. The array `UsedColors` is a boolean array that won't hold any necessary part of the solution — it is simply a transformation of `Coloring` to help state the objective.

In the objective function a sum expression is used. It sums all the elements in the `UsedColors` array together. Since the elements are either 0 or 1, this sum is the actual number of used colors.

Regarding the constraints, we see that there are two of them. The first one makes sure that two bordering countries don't have the same color. The second one states how the two array variables `UsedColors` and `Coloring` depend on each other.

Finally, there is a display statement that ensures that only colors that are used get displayed in the solution.

ESRA Now that I have explained how OPL works, I am ready to describe ESRA. As mentioned in the introduction of this report, ESRA is an extension of OPL. Most things that exist in OPL also exist in ESRA. Members of the ASTRA group had been using OPL for a long time and realized that it could be improved. They added new features to the OPL language and came up with a new language, which is ESRA. I will now describe what these new features are.

As mentioned above, OPL does not allow sets and ranges to be variables. However, while using OPL, members of the ASTRA group quite often used the concept of a variable set in their programs. As set variables are not allowed, they were forced to represent them as boolean arrays. With ESRA one can now declare set variables and even range variables. Here are some examples:

```
var {T} S;
var prefix(R) P;
var suffix(R) Sf;
var subrange(R) Sb;
```

Looking at the examples, the first one shows how to declare a set variable. The new set variable `S` will take values that are subsets of the existing set `T`. The three last examples show different ways of declaring a range variable. Prefix variable `P` represents a range at the beginning of the existing range `R`, suffix variable `Sf` at the end, and subrange variable `Sb` anywhere in the middle.

When declaring variables, like above, one calls the existing identifier, like `T` or `R`, the *domain* of the new variable. For example, the domain of `S` is `T`, the domain of `P` is `R`. The domain itself might also be a variable that has a domain. In this way, one gets whole chains, or more accurately trees, of domain dependencies. The identifiers at the root of these trees are called *ground*; the identifiers in the middle of the tree are called *non-ground*.

Probably the most important new feature in ESRA is mapping variables. They map values from one set, known as the *domain* of the mapping, to values in another set, known as the *codomain* of the mapping. Don't get this definition of domain mixed up with the definition of domain described in the paragraph above. They happen to have the same name but are completely different things. Here is how to declare a mapping variable:

```
var V->W M;
```

In this declaration, `M` is the new mapping variable being defined, `V` is the domain of the mapping and `W` is the codomain of the mapping. The nice thing about these mappings is that `V` and `W` don't have to be constants (ground). They can also be set and range variables (non-ground).

ESRA also introduces new operators to be used with the new types of variables. For range variables there are the operators `prefix`, `suffix` and `subrange`, which check if a range is a prefix, suffix or subrange respectively of another range. For mapping variables the functions `surjective`, `injective` and `bijective` are introduced, which check if these mathematical properties are true or not. Also some of the existing functions like `card`, `forall`, `sum`, etc. can now be used as well with the new types of variables.

Using the ESRA language and its new features, I can now rewrite the GCP program given in the paragraph about OPL. This shows the reader that using ESRA is a more efficient way of modelling a problem. Looking below, the reader can see, for example, that the second constraint from the OPL program is no longer needed — unlike OPL, the variable `UsedColors` can now be used in the declaration of `Coloring`.

```
enum Country ...;
enum Color ...;
struct border {
  Country c1;
  Country c2 };
{border} Borders = ...;
var {Color} UsedColors;
var Country -> UsedColors Coloring;
```

```
minimize
  card(UsedColors)
subject to {
  forall(B in Borders)
    Coloring.B.c2 <> Coloring.B.c1
};
```

See [6], section 3.2, for the complete grammar of ESRA.

Translating between ESRA and OPL Here, I just mention briefly the main principles used in translating ESRA into OPL. For a list of all the translation rules, see [6], section 4.2.

As mentioned above, the ASTRA group were using boolean arrays to represent their variable sets in OPL. This, of course, is how ESRA translates set variables into OPL: using boolean arrays.

Range variables can also be represented as boolean arrays in OPL. Constraints are added to the translation to ensure that all the 1s in the array are consecutive.

For mapping variables, different representations are used depending on whether the domain and the codomain are ground or non-ground. For example, if both are ground, only a simply array is needed. However, if both are non-ground, a boolean matrix plus two constraints and a display statement are required.

As with mapping variables, the translation of operators and functions in ESRA can be either simple or complicated. The membership operator, for example, is easily translated into one-line OPL statements, while the `sum` and `forall` primitives require the implementation of complex algorithms.

The simplified version of the ESRA language, and the OPL+ language When I implement the non-deterministic compiler in the second phase, I will ease my task by changing the input language from ESRA into a simplified version of ESRA, and changing the output language from OPL to a modified version of OPL, called OPL+. I describe the two languages already in this section, because an example in the next section uses the languages.

In the language referred to as the simplified version of ESRA many primitives have been removed. Only primitives that are necessary for creating ESRA models that can be compiled in a non-deterministic way are kept. For example, only mapping variables are used — set and range variables have been removed. See the user manual, listed as appendix A, for the complete grammar.

The modified version of OPL, OPL+, is a small extension of OPL. The only new feature is that also set variables are allowed, just like in ESRA. They are declared as normal: `var {T} S`. The reason for this small change is that it helps us to create more output language representations for the ESRA primitives. The non-deterministic compiler thus generates more interesting results.

2.5 Non-determinism in Compilation

The goal of the second phase of this project is to make the ESRA compiler non-deterministic. In this section I try to explain what non-determinism in compilation means.

Normally a compiler takes an input program and translates it into one output program. A non-deterministic compiler takes an input program and translates it into *several* output programs. The output programs accomplish the same task but are implemented in different ways. Put in terms of non-determinism, one says that the compiler generates a set of programs with different representations.

The purpose of generating different programs is that of efficiency in execution. Although they do the same thing, some programs might execute faster than other programs. By generating several programs the user can choose the program that executes the fastest.

The execution time can also depend on which input data is used with the program. One set of input data might be good for one program but bad for another, while on the other hand, another set of input data is bad for the first program, but good for the second one. By inputting to the non-deterministic compiler a typical set of input data that one is using for one's problem, the compiler is able to automatically test the input data with all the generated programs and select the fastest one.

Example on different representations Given the ESRA declaration `var V->W F`, where `V` denotes the domain and `W` the codomain, there are actually at least the following three different representations in the OPL+ language.

```
var V->W F;
1) var int F[V] in W;
2) var int F[V,W] in 0..1;
   forall(i in V)
     sum(j in W) F[i,j] = 1;
3) var {V} F[W];
   union all(j in W) F[j] = W;
   forall(i in W)
     forall(j in W)
       i <> j => F[i] inter F[j] = {};
```

The first representation is a 1-dimensional array, which is the most obvious way of representing a mapping.

The second representation is a 2-dimensional matrix, where the domain `V` represents the first dimension and the codomain `W` represents the second dimension. An element in the matrix is either 1 or 0, depending on whether the corresponding domain-codomain pair is a part of the mapping or not. The associated constraint ensures that every value in the domain maps to exactly one element in the codomain — this is the proper definition of a mapping.

The third representation defines an array of set variables where each value in the codomain is used to index its own set variable in the array. The set variable contains all the elements in the domain that map to this value in the codomain. The constraint ensures that an element in the domain doesn't occur in more than one set, again following the definition of a mapping.

See [?], chapter 4, for more examples showing non-deterministic rewrite rules.

Combining Representations Not yet mentioned is that an input model can contain *more than one* part that can generate different representations. Before I go on, let's henceforth denote parts of a model as *elements*. Regarding for example the ESRA language, there are input elements, output elements, constraint elements and objective elements, corresponding in turn to the data declarations, variable declarations, constraints and objective functions. For example, looking above at the second representation for the mapping example, we can see that an output element in the ESRA language is being translated into a set of two elements in the OPL language: one output element and one constraint element.

So expressed in other words, it is possible that several elements in the input model generates a set of different representations. This adds complexity to the generation of the models. A way to combine the representations is needed. The easiest way is to simply add the elements from the different representations to each other in a combinatory fashion. If, for example, there is one element in the input model generating two representations and another one generating three representations, the overall number of generated models is the product of two and three which is six.

Another thing that complicates matters is that representations generated from one element may depend on representations generated from other elements. Let's say, for example, that we have an input model consisting of the declaration of a mapping variable `F` and a constraint that contains a reference to `F`. Depending on which representation we use for `F` in the declaration, we need to be sure that we use the same representation for `F` in the constraint.

A complete example To sum things up, I give an example of an input program and all the models generated from it. First I need to introduce a new element and some of its representations.

```
injective(F):
1) alldifferent(F);
2) forall(i in V)
     forall(j in V) i <> j =>
       F[i] <> F[j];
3) var int D_F[W] in V;
   forall(i in V)
     forall(j in W)
       F[i] = j => D_F[j] = i;
4) forall(i in W)
     sum(i in V)
```

```
       F[i,j] <= 1;
5) forall(i in W)
     card(F[j]) <= 1;
```

Here we notice the dependency issue. The first three representations are to be used if `F` is using its first representation. In turn, representations four and five are required when `F` is represented as a matrix or an array of sets respectively.

Next I create the input program. It has three input elements: a set `V`, a set `W` and a mapping `F` from `V` to `W`; and one constraint element: `injective(F)`.

```
{int} V;
{int} W;
var V->W F;
solve {
  injective(F)
};
```

Using the representations for mappings and the injective constraint and applying the principle of combining representations described above, the following models are generated from the input program:

```
1)
{int} V;
{int} W;
var F[V] in W;
solve {
  alldifferent(F)
};
2)
{int} V;
{int} W;
var F[V] in W;
solve {
  forall(i in W)
    forall(j in W)
      i <> j => F[i] <> F[j]
};
3)
{int} V;
{int} W;
var F[V] in W;
var D_F[W] in V;
solve {
  forall(i in V)
    forall(j in W)
      F[i] = j => D_F[j] = i
};
```

```
4)
{int} V;
{int} W;
var F[V,W] in 0..1;
solve {
  forall(j in W)
    sum(i in V) F[i,j] = 1;
  forall(i in W)
    sum(i in V)
      F[i,j] <= 1 };
5)
{int} V;
{int} W;
var {V} F[W];
solve {
  union all(j in W)
    F[j] = W;
  forall(i in W)
    forall(j in W)
      i <> j => F[i] inter F[j] = {};
  forall(i in W) card(F[j]) <= 1
};
```

Chapter 3

Phase I: The ESRA Compiler

3.1 Goals and Requirements

The primary goal of phase 1 is to create the ESRA compiler. The ESRA compiler should take as input a program written in the ESRA language and return as output the corresponding program written in the OPL language. The grammar of ESRA can be found in [6], section 3.2; the grammar of OPL in [5]; and the ESRA-to-OPL rewrite rules in [6], section 4.2.

For some parts of the ESRA language the translation rules have not been created yet. The sum and forall expressions have translation rules given in the form of algorithms. I need to implement these. For the range operators neither algorithms nor translation rules exist. These also need to be implemented.

Another requirement is that the compiler should be able to report an error when there is something wrong with the ESRA input program. The error message should be informative in such a way that it is easy to locate the error and correct it.

For the compiler to be easy to use, there should also be a graphical interface. It should allow the user to create, open and save files, as well as compile files.

The final requirement is that it should be easy to change the grammar and the translation rules. One shouldn't have to change things in several places in the compiler code.

3.2 Solution and Method

As programming language I have chosen Java. Java is faster than Prolog and more user friendly than C.

I use two good help tools for creating the compiler: JLex and JavaCUP. JLex is for creating the tokenizer and JavaCUP is for creating the parser. Both work similarly to the tools lex and bison for C.

From the ESRA grammar I have created both a JLex specification file and a JavaCUP specification file. Since the format of the grammar in [6] was not the

same as the format of the grammar that JavaCUP accepts, I needed to convert the grammar in [6], see section 3.6.

From these specification files, a Java tokenizer class and a Java parser class are generated. The Java tokenizer produces tokens that are input to the Java parser. For this I have investigated and solved the problem of how the two tools can be combined, see section 3.3.

By analyzing the rewrite rules, I see that I'm going to need a two-pass compiler. This is because most of the rules are dependent on knowing what type the identifiers in the expressions have. In the first pass I create a symbol table of all the identifiers, and in the second pass I do lookups of the identifiers in the symbol table. To see how I use JavaCUP to create a two-pass compiler, see section 3.4.

I have also incorporated an error detection system into the compiler. The system is able to detect in which step of the compilation process the error occurs. If the error occurs in the tokenizer, there is a lexical error; if the error occurs in the parser, there is a syntax error; and if the error occurs in the translator, there is a semantic error. The system also reports which token caused the error, in what row, and in which column. To see how I did this with JLex and JavaCUP, check section 3.5.

To ease my work of developing the compiler, I have developed two tools in Perl: the token list generator generator, see section 3.8, and the parse tree generator generator, see section 3.9. The two tools help display what the list of tokens and the parse tree look like after the tokenizing step and the parsing step respectively. They are useful when an error occurs while constructing the compiler, because they help me determine in which step of the compilation process the error started. For example, if there is a parse error, the error could be either that there is something wrong in my parser or that the tokenizer generated an incorrect list of tokens. By inspecting the list of tokens I am able to check this.

I have also completed the missing translation rules. See section 3.7 on how I created the rules for the range operators in the ESRA language.

For the requirement of that it should be easy to write and change the translation rules, I have created a rule file system, see section 3.10. One writes the rules in a normal text file in almost the same way as they are written in [6]. Then I have written a special rule converter program in Perl that converts the rule file with the rules into a Java program with corresponding Java methods.

It is also easy to modify the grammar. By using JavaCUP the grammar is specified in the JavaCUP specification file. The format of the grammar used in the file is almost identical to the standard format used in ASTRA reports. Almost the only difference is that `::=` is used instead of an arrow. To modify the grammar one simply edits the specification file.

Finally I have also created the ESRA user interface, see section 3.11.

3.3 Combining JLex and JavaCUP

For writing the ESRA compiler, I use both JLex and JavaCUP. My problem is therefore to figure out how to put JLex and JavaCUP together. The documentation on how to connect JLex to JavaCUP is very poor, both in the JLex manual and in the JavaCUP manual, as well as on the Internet.[1] I solved the problem by using JLex and creating a tokenizer for the example parser in the JavaCUP manual. I discovered that one needs to do the following things in one's JLex specification to make it compatible with one's JavaCUP specification.

First, one needs to add the `%cup` directive. This is equal to three other directives: `%type Symbol`, `%function next_token`, and `%implements java_cup.runtime.Scanner`. They alter the .lex.java file that is produced. By default, the definition of the `Yylex` class in the .lex.java file won't implement anything. In turn, the method in the `Yylex` class for reading tokens gets called `yylex` and returns `Yytoken` objects. With these directives the `Yylex` class will implement `java_cup.runtime.Scanner`. In turn, the method gets called `next_token` and returns `Symbol` objects. All these changes are needed so that the parser class can access the tokenizer class.

Second, one needs to change one's semantic actions that are associated with the token definitions, so that they include code that create and return `Symbol` objects. The `Symbol` constructor is called with an integer denoting the terminal it represents and an optional value of type `Object`. The thing is — when JavaCUP generates the parser class it also generates a class called `sym`. This class contains all the terminals represented as integer constants. By using these constants, JLex can tell JavaCUP which terminals the tokens stand for.

Here are two examples using the `Symbol` constructor:

```
<YYINITIAL> "+" {
  return new Symbol(sym.PLUS);
}
<YYINITIAL> {NUMBER} {
  return new Symbol(
    sym.NUMBER,new Integer(yytext());
}
```

There is an important side note related to the above paragraph. As stated, JLex uses the constants in the `sym` class. This means that one must generate the parser before one generates and compiles the tokenizer. This is contrary to the intuitive order in which one would generate and compile the two.

Third, one also needs to include the `%eofval` directive as follows:

```
%eofval{
  return new Symbol(sym.EOF);
%eofval}
```

[1] Today, six months later, an example showing how to combine JLex and JavaCUP has been added to the JavaCUP home page.

If one doesn't have have this directive, the parser runs and produces the correct result, but never terminates. The `EOF` constant is a constant that JavaCUP includes by default in the sym file.

The fourth and final thing that one needs to do is add an `import` statement in the user code section:

```
import java_cup.runtime.*;
```

This is because the `Symbol` class that one is using resides in the `java_cup.runtime` package.

Finally, here is what the whole file looks like.

```
import java_cup.runtime.*;
%%
%cup
DIGIT=[0-9]
NUMBER=({DIGIT})*
WHITE_SPACE=([\ \n\r\t\f])+
%eofval{
  return new Symbol(sym.EOF);
%eofval}
%%
<YYINITIAL> "+" {
  return new Symbol(sym.PLUS);
}
...
<YYINITIAL> {NUMBER} {
  return new Symbol(
    sym.NUMBER,new Integer(yytext());
}
```

3.4 Making a Two-pass Compiler with JavaCUP

For compilations of some languages, it is impossible to complete the translation of the code in one pass. These languages require a two-pass parser. The ESRA language is such a language. It requires the code to be stepped through twice, because several of the ESRA-to-OPL rewrite rules need to know the types of the identifiers. In the first pass the identifiers and their types are stored in a symbol table, and in the second pass the symbol table is used to make the actual translation. The problem is therefore to figure out how to use JavaCUP to create a two-pass parser.

The Toy Language I solved this problem by inventing a new language called the Toy language. The Toy language has certain features that require one to pass

through the code twice. However, it is much smaller than the ESRA language. In this way, I have isolated the problem to a much smaller space. If I can create a two-pass parser for the Toy language, I will also be able to create it for the ESRA language.

A little simplified, the Toy language consists of a series of declarations on separate lines separated by semi-colons. Every declaration can be either a ground or a non-ground declaration. Here is a piece of the grammar that we will focus on:

```
<Declarations> -> <Declaration> ;
                | <Declaration> ;<Declarations>
<Declaration>  -> ground <Id>
                | non_ground <Id> <Id>
```

These ground and non-ground declarations build up a set of hierarchical tree structures, where identifiers declared as ground are root elements of these trees, and identifiers declared as non-ground are nodes in the tree below the root. In the case of `non_ground <X> <Y>`, the identifier `X` is the parent node of `Y`. The domain of an identifier, `dom(id)`, is the root of the tree that the identifier exists in.

In the translation process, the non-ground declaration is translated in two different ways, depending on whether the parent identifier is ground or non-ground. Here is the translation rule for the non-ground declaration:

```
non_ground X Y ==> var int Y[X] in 0..1
                 | if X is ground

               ==> var int Y[dom(X)] in 0..1
                 | if X is non-ground
```

This is the reason why we need two passes of the code. In the first pass we parse the declarations and store which identifiers are ground and which are non-ground. In the second pass we use this stored information to translate the non-ground declarations properly.

To create this two-pass parser I created two JavaCUP specification files: toy_pass1.cup and toy_pass2.cup. They are the specifications for the first and second pass parsers respectively. I also created a Java class called `DomainTree` for storing the first-pass information. Finally I have a main class called `toy` that creates and runs the two parsers and connects them together.

The First Pass It is in the first specification file that I construct the `DomainTree` object. I do this by using the `action code` directive.

```
action code {:
  DomainTree domainTree = new DomainTree();
:}
```

Then I use the two methods of the `DomainTree` class, `addGround` and `addNonGround`, to build up the tree:

```
Declaration ::= GROUND IDENT:id
  {: domainTree.addGround(id); :}

              | NON_GROUND IDENT:pid IDENT:id
  {: domainTree.addNonGround(pid,id); :}
```

Finally, we need to get the translated result back to the caller that invoked the parse. For this we need to do two things. First we need to declare that the starting non-terminal is of type `DomainTree`:

```
non terminal DomainTree Declarations;
```

Then we need to let the semantic code for the starting non-terminal return the `DomainTree` object:

```
Declarations ::= Declaration SEMI
  {: RESULT = domainTree; :}

               | Declaration SEMI Declarations
  {: RESULT = domainTree; :}
```

The Second Pass In the second pass we use the information gathered from the first pass. The big problem is how to actually transfer that information. The problem is actually two-fold. This is because the parser that is generated is divided into two classes: the parser class and the actions class. Since the `DomainTree` object is used by the semantic code, I first need to get the `DomainTree` object into the parser class, and then from there into the actions class.

In JavaCUP there is a directive for extending the parser with custom variable and method declarations. I take advantage of this in the second specification file by creating a second constructor to the parser class. It takes a second argument which is the `DomainTree` object. I also add a new variable `domainTree` which stores the `DomainTree` object that is submitted through the constructor:

```
parser code {:
  DomainTree domainTree;
  public toy_parser_pass2(
    java_cup.runtime.Scanner s,
    DomainTree domainTree)
  {
    this(s);
    this.domainTree = domainTree;
  }
}
```

To get the `DomainTree` object from the parser class into the actions class I use the `action code` directive. It is similar to the `parser code` directive but is used for extending the actions class. I create a method for setting the `DomainTree` object in the class. I also add a member variable, `domainTree`, that serves as a place holder for the `DomainTree` object in the actions class:

```
action code {:
  DomainTree domainTree;
  public void setDomainTree(
    DomainTree domainTree)
  {
    this.domainTree = domainTree;
  }
:}
```

I now need to have the parser class call the method above before it starts executing the semantic code. I solve this by placing the method call in the `init with` directive. However, I need to have a reference to the actions class object from within the parser class. By inspecting the parent class of the parser class, `lr_parser` in the `java_cup.runtime` package, in which the `parse` method is defined, I find that I can use the variable `action_obj` as a reference:

```
init with {:
  action_obj.setDomainTree(domainTree);
:}
```

Putting It All Together By using JavaCUP, I generate the two Java parser files for the Toy language. By default the generated Java file is called parser.java. Since we cannot have two Java classes with the same name, I need a way to generate the classes with different names. The solution is to use a JavaCUP command-line option called `-parser`. By doing this, I create the two files toy_parser_file1.java and toy_parser_pass2.java:

```
java java_cup.Main -parser toy_parser_pass1
  < toy_pass1.cup
java java_cup.Main -parser toy_parser_pass2
  < toy_pass2.cup
```

There is also a `-symbols` option used for specifying a different name for the `sym` class. However, since the two parsers parse the same language, they may use the same `sym` class.

To put it all together I create the Java file `toy.java`. It creates the two parser objects and connects them together. The whole file is shown below. It reads the input code from standard input and prints the translation result to standard output. Note that a problem was that I needed to read the same input stream twice — one time for the first parse and a second time for the second parse. As seen in the code, I solved this by using the methods `mark` and `reset`.

I used `mark` to mark the stream at the beginning of the first pass, and I used `reset` to reset the stream to that point at the beginning of the second pass. Note that the argument to `mark` is how many bytes that can be read before the mark becomes invalid. Since I never want it to become invalid, I make this argument very big.

```
InputStream stream_obj = System.in;
stream_obj.mark(100000);
Yylex lexer_obj = new Yylex(stream_obj);
toy_parser_pass1 parser_obj = new
  toy_parser_pass1(lexer_obj);
DomainTree domainTree =
  (DomainTree) (parser_obj.parse().value);
stream_obj.reset();
lexer_obj = new Yylex(stream_obj);
toy_parser_pass2 parser_obj2 = new
  toy_parser_pass2(lexer_obj,domainTree);
String result =
  (String) (parser_obj2.parse().value);
System.out.print(result);
```

3.5 Reporting Errors with JLex and JavaCUP

When the user writes his ESRA program it will sometimes contain errors. Perhaps he has forgotten a semi-colon; perhaps he has forgotten to declare a variable he is using. As with any other programming language, we would like our compiler to catch these errors and report them. The more detailed the descriptions of these errors, the greater the chances of the user finding and correcting them.

By default, JLex and JavaCUP return very vague error descriptions. In JLex when there is an error, it throws an `Error` object with the message: "`Lexical error: Unmatched input`". In JavaCUP when there is an error, it throws an `Exception` object with the message: "`Can't recover from previous error(s)`". We would like these error descriptions to be more informative; for example, we would like them to contain information about which token caused the error, on which line the error occurred, and at which character position the error occurred.

JLex To solve this problem in JLex I add the following line to the bottom of the JLex specification file:

```
. { throw new TokenizerError(yytext(),yyline,yychar); }
```

By doing this, the JLex own `Error` object will never be thrown. Instead, all erroneous tokens are matched by the dot, and our own defined `TokenizerError` is therefore always thrown. As we are now throwing an `Error`, we need to have the `yylex` method declare this. We do this by adding the following to the JLex directives section:

```
%yylexthrow{
  TokenizingError
%yylexthrow}
```

The entities `yyline` and `yychar` are defined entities in JLex which refer to the current line and the current character position. One needs to invoke these entities by adding the following to the JLex directives section:

```
%line
%char
```

Note that these entities are zero-based, not one-based; that means that `yyline` and `yychar` are 0 for the first row and column respectively, 1 for the second row and column respectively, and so on. Note also that `yychar` refers to the character position of the entire text. If one wants the character position of just the current line, one can do the following trick: first one defines a new variable called `newline_pos`:

```
%{
  int newline_pos = 0;
%}
```

Every time there is a new line, one updates the `newline_pos`. It therefore always refers to the character position of the most recent new line character:

```
<YYINITIAL> "\n" {
  newline_pos = yychar;
}
```

Then finally we get the character position on the current line by simply subtracting `newline_pos` from `yychar`:

```
. { throw new TokenizerError(yytext(),yyline,yychar-newline_pos); }
```

JavaCUP By default, when there is a parse error, JavaCUP gives an `Exception` with the message: "`Can't recover from previous error(s)`". As with the JLex error, we would like it to give us the token, line and column so that we more easily can locate and correct the error.

The generated parser class inherits from the `lr_parser` class in the `java_cup.runtime` package. In it there is an error handling routine that gets called by the parser when errors occur. Its name is `report_fatal_error` and is the method that actually throws the exception. By overriding this method in the generated parser class, we ourselves can control how the error handling should work.

The remaining problem concerns how the error handling method can get access to the line, row and column information of the erroneous symbol. We make use of the fact that the erroneous symbol is passed as an argument to the `report_fatal_error` method. It seems suitable therefore to pack the information in the symbol object. Recall that a symbol object has two entities:

the token identifier and a `value` object. Normally, one uses the `value` object to store the `String` object of an identifier or the `Integer` object of a number. We create a new class, `SymbolInfo`, that holds the entities `line`, `column`, `token` and `value`. As we generate the tokens in the lex file we also create and add a `SymbolInfo` object to every `Symbol` object that we generate:

```
<YYINITIAL> "forall" {
  return new Symbol(sym.FORALL,new SymbolInfo(yytext(),yyline,yychar));
}
```

Note that the `SymbolInfo` can also take a fourth argument which is the `value` entity. It is used, for example, by identifiers and numbers that no longer can use the `value` entity of the `Symbol` object.

We finish by creating the `report_fatal_error` method. It simply unpacks the `SymbolInfo` object and throws a custom-made parse error with its token, line, and column information:

```
parser code {:
  public void report_fatal_error(
    String   message,
    Object   object)
    throws ParsingError  {
    Symbol symbol = (Symbol) object;
    SymbolInfo info = (SymbolInfo) symbol.value;
    done_parsing();
    throw new ParsingError(info.token,info.line,info.column);
  }
:}
```

3.6 Converting the Grammar to JavaCUP-format

In JavaCUP, one can only write the production rules of the grammar using a very simple format, see [2]. In [6], section 3.2, the grammar of the ESRA language is written using a more advanced format. The difference lies in how one can specify each production alternative: JavaCUP only allows a sequence of terminals and non-terminals, while the conventions used in [6] also allow the following syntax:

- [*foo*] — *foo* is optional
- {*foo*} — zero, one, or several times *foo*
- *foo*+ — one or several times *foo*, separated by commas.
- *foo** — one or several times *foo*, separated by semi-colons.

I therefore need to find a way to rewrite the grammar in [6] into the simpler format allowed by JavaCUP. I construct and use the conversion rules that the reader can see below. For each conversion rule, grammar rules that match the left-hand side of the rule (the side that is to the left of the arrow, ==>), should be replaced by the corresponding rules on the right-hand side of the rule. The entities `A`, `B`, `C`, and `D` that occur in the left-hand side of the rule match any arbitrary sequence of terminals and non-terminals. The values that they're assigned to are copied into the entities with the same names in the right-hand side of the rule. The occurrence of the entity `E`, also in the right-hand side of the rules, represents a new non-terminal, which must be created.

```
A -> B[C]D  ==>  A -> BD
                   -> BCD

A -> B{C}D  ==>  A -> BD
                   -> BED
                 E -> CE
                   -> C
A -> BC+D   ==>  A -> BD
                   -> BED
                 E -> ,CE
                   -> ,E
A -> BC*D   ==>  A -> BD
                   -> BED
                 E -> ;CE
                   -> ;E
```

Below, I give an example that shows the use of the conversion rules in practice. Here, first, is a part of the grammar in [6]. It is the grammar rule for the `Model` non-terminal. Entities inside < and > are non-terminals.

```
<Model>  ->  {<Declaration>}
             <Instruction>
```

This grammar rule matches the left-hand side of the *second* conversion rule of those listed above. The `Model` non-terminal matches the entity `A`, the `Declaration` non-terminal matches the entity `C`, and the `Instruction` non-terminal matches the entity `D` (the entity `B` is matched to the empty string). Applying the conversion rule, the following new rules are created, see below. Notice the creation of the new non-terminal `Declarations`, which corresponds to `E`.

```
Model         ->  Instruction
                |  Declarations Instruction
Declarations  ->  Declaration
                |  Declaration Declarations;
```

3.7 Writing the Range Operator Rules

One of the requirements of this project was to create the translation rules for the range operators in the ESRA language. The range operators are `prefix`, `suffix` and `subrange`. They all take two arguments and enforce that the first argument is a prefix, suffix or subrange of the second argument. What makes this task complex is that each argument can be either a range or a range variable. This means that for each range operator there are four different cases, and all in all I have to create twelve different translation rules.

Several problems arose during my work in trying to create these translation rules. First, there was the problem of how to treat empty range variables; since a range variable is represented as a boolean array, it would be possible that all entries were zero. Second, a lot of the constraints that I constructed consisted of very long logical expressions involving *logical or* and *implication*, which would take a lot of time for the solver to solve. Lastly, I realized that OPL uses strict evaluation, so a lot of simple solutions that were based on OPL not evaluating the second argument had to be thrown away, since they caused *index out of bounds errors* for the arrays.

Due to these problems the requirements were simplified. Each operator could now only take range variables as arguments and these range variables had to be of the same domain, i.e., their parent ranges had to have the same lower and upper bounds.

Before I present the solution I need to discuss how range variables in ESRA are represented in OPL. Below is shown how a range variable in ESRA can be declared in three different ways. `S` will be the new range variable and `R` its domain range.

```
var prefix(R) S;
var suffix(R) S;
var subrange(R) S;
```

Their representations in OPL all have the following boolean array:

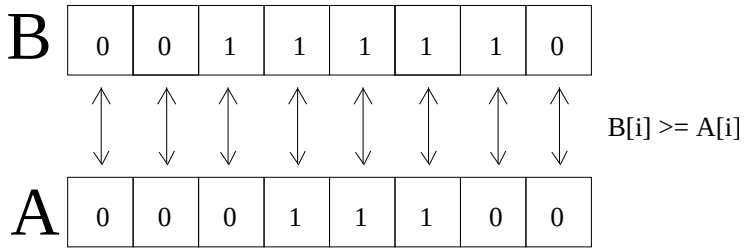
```
var int S[R] in 0..1;
```

They also all have a constraint that ensures that all `1`s in the array are contiguous, i.e., there are no `1`s in the array that are separated by any `0`s. The way to interpret this representation is that all elements that are `1` are part of the range and all element that are `0` are not part of the range. The index of the left-most `1` is the lower bound of the range and the index of the right-most `1` is the upper bound of the range.

Using the above information about range variables I can now create the translation rules for the range operators. When discussing the solution of each operator, I will use the term *interpreted range* of an array to mean the contiguous stretch of `1`s that occur in the array.

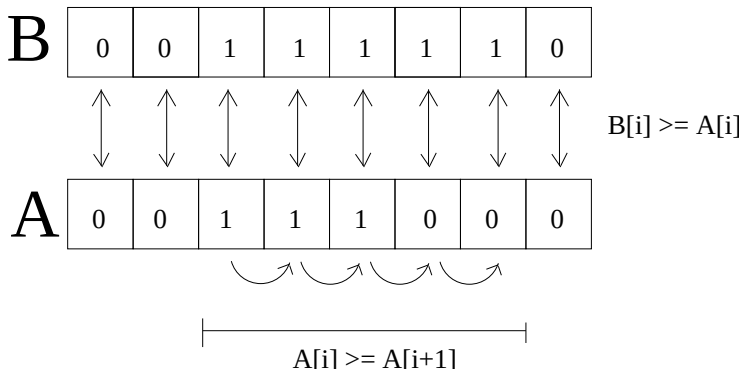
For the `subrange` operator I create a constraint that checks that every element in `B` is greater or equal to the element at the same index in `A`. This means

that for all 1s that occur in A there should be 1s at the same indices in B, making the interpreted range of A be inside the interpreted range of B. The figure below shows two possible instances of the arrays A and B, such that the interpreted range of A is a subrange of the interpreted range of B. The arrows show the comparisons of the elements made by the OPL code. Below the figure the OPL code is listed.




```
subrange(A,B);
  =>  forall(i in [a..b])
      B[i] >= A[i];
    |  A and B are range variables
       with the same domain a..b
```

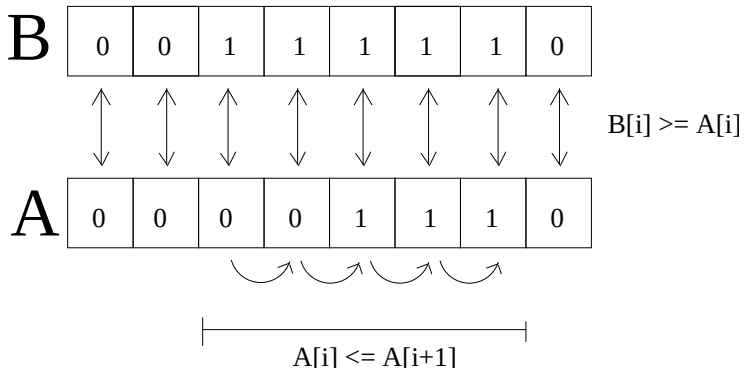
For the prefix operator I use two constraints. The first one is the same as the one used for the subrange operator. The second one checks that every pair of adjacent elements in A that are within the interpreted range of B, are such that the first element is greater or equal to the second element. This means that for the left most 1 in B, there must be a 1 at the same index in A. Therefore the interpreted range of A is a prefix of B. The purpose of the if construct is to prevent that an *array index out of bounds* error occurs if a is greater or equal to b.




```
prefix(A,B);
  =>  forall(i in [a..b])
```

```
      B[i] >= A[i];
    if b > a then
    forall(i in [a..b-1])
      B[i] = 1 & B[i+1] = 1 =>
        A[i] >= A[i+1]
  |  A and B are range variables
    with the same domain a..b
```

For the `suffix` operator I use the same two constraints as for the `prefix` operator. The only change I do is to reverse the greater or equal sign into a less or equal sign in the second constraint. Using the same reasoning as for the `prefix` operator above, the interpreted range of `A` must now be a suffix of `B`.




```
suffix(A,B);
  =>  forall(i in [a..b])
        B[i] >= A[i];
      if b > a then
        forall(i in [a..b-1])
          B[i] = 1 & B[i+1] = 1 =>
            A[i] <= A[i+1]
    |  A and B are range variables
      with the same domain a..b
```

3.8 Token List Generator Generator

Creating the compiler can be a tricky job. If something goes wrong and the wrong code is produced, it can be hard to find the bug. To help with this, I have created two helper programs: *token list generator generator* and *parse tree generator generator*.

Token list generator generator is a Perl program that takes the lex file and transforms it into a new lex file. This new lex file, when run, instead of returning symbols to the parser, prints out a nice and readable list of the tokens produced from the input file. I now give an example.

Here is a simple ESRA program:

```
int x;
int y;
solve x = y;
```

And here is the print-out one gets from running it through the token list generator generator:

```
[INT:'int']
[ID:'x']
[SEMI:';']

[INT:'int']
[ID:'y']
[SEMI:';']

[SOLVE:'solve']
[ID:'x']
[EQUAL:'=']
[ID:'y']
[SEMI:';']
```

3.9 Parse Tree Generator Generator

Parse tree generator generator works similarly to the token list generator generator. It generates a new cup file from the old one which, when run on an input file, displays the parse tree of that input file.

By using these programs, I can see what's going on in the compilation process and narrow the location of the error down to one of the following: the tokenization, the parsing, or the translation. Here follows an example of the printout that is produced from the parse tree generator generator. It is based on the simple ESRA program that is listed in the previous section.

```
Model {
  Declarations {
    Declaration {
      DataDecl {
        Type {
          INT
        }
        ID:x
      }
      SEMI
    }
    Declarations {
```

```
      Declaration {
        DataDecl {
          Type {
            INT
          }
          ID:y
        }
        SEMI
      }
    }
  }
  Instruction {
    SOLVE
    Constraint {
      Formula {
        Atom {
          Expression {
            Argument {
              ID:x
            }
          }
          ArithOp {
            EQUAL
          }
          Expression {
            Argument {
              ID:y
            }
          }
        }
      }
    }
    SEMI
  }
}
```

3.10 Rule Converter

One big part of my job was to implement the ESRA-to-OPL translation rules. These rules are all listed in the report *The Syntax and Semantics of ESRA*, see [6]. The following example shows the format of such a rule in the report. This particular example is the rule for the suffix variable declaration.

```
var suffix(R) S;
   =>   var int S[R] in 0..1;
```

```
        forall(I in [L..U-1])
          S[I] <= S[I+1];
    |   R is a range L..U
```

Looking at the rule, the first line is the ESRA statement that is being translated. `R` and `S` are called the *parameters* of the rule. The parameters are unbound entities that will be matched to values during the translation. Then, all the occurrences of the parameters in the body of the rule will be substituted with the values. Regarding the body of the rule, it consists of a clause starting with an arrow, called the *output part,* and a clause starting with a vertical bar, called the *condition part.* There can be several such pairs consisting of an output part and a condition part in the body of the rule; these are called *subrules.* The translated result corresponds to the output part of the first subrule of which the condition part matches. Note in the example how the condition part is being used to extract the lower and upper bounds, `L` and `U`, of the domain range, `R`, so that these can be used in the output part.

Implementing the ESRA-to-OPL translation rules meant writing them in Java so that they could be used in the compiler. Shown below is the Java code that I had to write for the suffix variable declaration rule. Like it and all other rules, they are implemented as methods of which the arguments are the parameters of the rule.

```
public String suffixVarDecl(String P,String R)
    throws UndefinedIdentifierException,
           UnsatisfiedRuleException {
    String result = "";
    {
        SymbolData RData = symbolTable.lookup(R);
        if (RData == null)
            throw new UndefinedIdentifierException(R);
        String L = RData.rangeLoValue();
        String U = RData.rangeHiValue();
        if ((RData.isRange())) {
            result += (tab("var int "+P+"["+R+"] in 0..1",0));
            constraint(tab("forall(I in ["+L+".."+U+"-1]) \n",0) +
            tab(""+P+"[I] >= "+P+"[I+1]",1));
            return result;
        }
    }
    throw new UnsatisfiedRuleException();
}
```

I noticed that writing these methods by hand into the computer had many disadvantages. For example, constructing the Java strings for the output part was very tedious: I had to substitute all parameters with pluses (+), quotation marks ("), and the corresponding variable names in Java. Also, the methods,

in their structure, were very similar to each other, and I felt I was repeating the same job over and over again.

I therefore have implemented a Perl program that I call the *rule converter*. The rule converter lets one write translation rules in a file called *the rule file*. It then converts these rules automatically into Java methods, the same Java methods as were created by hand, as the example above showed.

The format of the translation rules in the rule file is very similar to that of the translation rules in the report [6], of which one is given in the beginning of this section. The exact format of the rule is as follows. First comes the *rule type*, which can be one of `decl`, `cons` or `expr`. `decl` is used if the statement being translated is a declaration; `cons` is used if the statement being translated is a constraint; and `expr` is used if the the statement being translated is an expression. Following the rule type is the *rule name*, which will be the name of the corresponding Java method. After the rule name comes the parameters of the rule enclosed in the parentheses, which will be the arguments of the corresponding Java method. Lastly follows the body of the rule enclosed within `{:` and `:}`. Here, the output parts are specified exactly the same as in the report [6]. The difference lies in the condition parts, where being formal is required. The condition part is a semi-colon separated list of conditions. Each condition consists of the sequence: an identifier, a colon (`:`), and a comma separated list of ESRA types, which the identifier is allowed to have. The ESRA types are `int`, `array`, `enum`, `range(L..U)`, `setvar`, `rangevar`, and `mapping(V->W)`. As can be seen, the `range` type and the `mapping` type allow one to extract entities, which can be used in the output part. `range` lets one extract the upper and lower bounds of the range (`L..U`), while `mapping` lets one extract the domain and the codomain of the mapping (`V->W`).

The example below illustrates the format of the translation rules, by showing how one may write the translation rule for the suffix variable declaration.

```
decl suffixVarDecl(R,S) {:
   =>   var int S[R] in 0..1;
        forall(I in [L..U-1])
          S[I] <= S[I+1];
    |   R:range(L..U)
:}
```

Besides sparing one the trouble of writing several quotation marks (“) and pluses (+), and letting one extract entities from range and mapping identifiers, here are other important benefits with the rule file:

- One may write such expressions as `dom(S)` in the output part. The expression gets substituted with Java code that evaluates to the domain of `S`. For example, here is part of the rule for the mapping variable declaration, which is given in the report [6]. Notice all the occurrences of `dom(V)` and `dom(W)` in the rule. The rule will be converted correctly into Java.

  ```
  var V->W M;
  ```

```
=>   var int M[dom(V),dom(W)] in 0..1;
     forall(I in dom(V)) sum(J in dom(W))
       M[I,J] = V[I];
     forall(J in dom(W),I in dom(V))
       M[I,J] <= W[J];
     display(I in dom(V),J in dom(W):
       M[I,J]=1) <I,J>;
 |   V:setvar,rangevar;W:setvar,rangevar
```

- Moreover, the rule converter detects if a line is a display, a declaration, a constraint, or an expression statement. Depending on the type of statement, the rule converter creates the appropriate Java code, so that the statement is placed correctly in the generated OPL code.
- The rule converter also detects indentions among the lines in the output part and generate corresponding calls to a Java tabbing function. In this way, the indentions used in the rule file get preserved in the generated OPL code.

3.11 Other Issues

In this section I mention some things from this phase which are of less importance, but nevertheless took pretty much time of the work.

First of all, there is the making of the graphical interface. The interface combines all the functionality of the ESRA program. It lets one view the list of tokens, the parse tree, the symbol table, and the final translation. It also provides the possibility to open and save files like a normal text editor. It even gives all the classical warnings like "`Do you want to save this file before closing it`", "`This file already exists, do you want to overwrite it`", etc.

Another significant part of this phase was implementing the translation algorithms for the forall constraint and the sum expression. The algorithms are listed in [7].

3.12 Testing and Results

In this section I do tests to show that the ESRA implementation works correctly. In section 2.4, about ESRA and OPL, I talked about a classical constraint problem called the Graph Coloring problem, GCP. Here I use two other classical problems called the *Warehouse Location problem*, *WLP*, and the *File Packing problem*, *FPP*, to test my program.

The WLP is about mapping stores to warehouses such that a certain cost is minimized. The cost is dependent on the number of warehouses open and the supply costs for these warehouses. The supply cost is specified individually for all the possible pairs of warehouses and stores in the problem. The main

constraint in the problem is that each warehouse has a maximum capacity of stores it can supply.

The FPP maps files to diskettes, such that a minimum number of diskettes are used. The files all have different sizes and cannot be broken apart. The aim is thus to place them in such a way that the non-used space on the diskettes is minimized. For more details on this problem and the WLP, see [7].

Now I use each problem (the WLP and the FPP) to test my application using the following procedure.

- Step 1: Create an ESRA model of the problem.
- Step 2: From the ESRA model generate an OPL program using my ESRA application.
- Step 3: Compare the generated OPL program to a handwritten OPL program for the same problem.
- Step 4: Create input data for the generated OPL program to be used with the *OPL constraint solver application*. This application, made by ILOG, is an interactive environment for designing and solving OPL programs.
- Step 5: Generate the solution to the generated OPL program with the input data using the OPL constraint solver application.
- Step 6: Solve the problem on paper using the same input data and compare it to the generated solution.

Testing with the WLP In this section I test my ESRA application with the WLP using the procedure described above.

For step 1, I use the ESRA model from [7]. The only modifications here are that, firstly parentheses are added to the cost function; the model would not compile correctly otherwise, and secondly the constraint `max-image-capacity` is used; the model in the report uses a `forall` constraint and a `count` expression to achieve the same thing.

```
int MaintCost = ...;
int NbStores = ...;
enum Warehouses ...;
range Stores 0..NbStores-1;
int Capacity[Warehouses] = ...;
int SupplyCost[Stores,Warehouses] = ...;
var {Warehouses} OpenWarehouses;
var Stores->OpenWarehouses Supplier;
minimize (sum(I->J in Supplier)
  SupplyCost[I,J]) +
  (card(OpenWarehouses) * MaintCost)
subject to {
  max-image-capacity(Supplier,Capacity)
};
```

For step 2, the following OPL program is generated from the ESRA model using my ESRA application. By brief inspection, the OPL program seems to have no obvious errors.

```
int MaintCost = ...;
int NbStores = ...;
enum Warehouses ...;
range Stores 0..NbStores-1;
int Capacity[Warehouses] = ...;
int SupplyCost[Stores,Warehouses] = ...;
var int OpenWarehouses[Warehouses] in 0..1;
var Warehouses Supplier[Stores];
minimize (sum(I in Stores)
  (SupplyCost[I,Supplier[I]]))+
  (sum(I in Warehouses)
  OpenWarehouses[I]*MaintCost)
subject to {
  forall(J in Warehouses)
    OpenWarehouses[J]=1 => sum(I in Stores)
    (Supplier[I]=J) <= Capacity[J];
  forall(I in Stores)
    OpenWarehouses[Supplier[I]]=1;
};
display(I in Warehouses:
  OpenWarehouses[I]=1) <I>;
```

For step 3, I compare the OPL program generated by the ESRA application with the OPL program generated by hand in the thesis [7]. They are semantically equal. The only syntactical differences are that they use different names for some of the iterating identifiers, and that the OPL program here uses more parentheses which is a result of the parentheses introduced in step 1.

For step 4, I create the following input data to be used with the OPL program in the OPL constraint solver application. The values have been chosen in such a way that finding the solution is not straight forward.

```
MaintCost = 10;
NbStores = 8;
Warehouses =
{uppsala,stockholm,vasteras,linkoping};
Capacity = #[
  uppsala:3,
  stockholm:3,
  vasteras:1,
  linkoping:4
]#;
SupplyCost = #[
  0: [5,8,4,3],
```

```
  1: [3,9,2,2],
  2: [4,7,1,5],
  3: [6,3,5,4],
  4: [2,3,6,1],
  5: [3,4,9,3],
  6: [3,5,5,3],
  7: [2,5,1,4]
]#;
```

For step 5, the following solution is generated by the OPL constraint solver. By brief inspection, I see that the cost value has been calculated correctly, and that the maximum capacity constraint is fulfilled. What remains is testing if the cost value is the actual minimal cost value.

```
objective value: 49
variable OpenWarehouses:
uppsala    - 1
stockholm  - 0
vasteras   - 1
linkoping  - 1
variable Supplier:
0 - linkoping
1 - linkoping
2 - vasteras
3 - linkoping
4 - linkoping
5 - uppsala
6 - uppsala
7 - uppsala
```

For the final step, step 6, I will present handwritten proof that the solution generated by the OPL constraint solver application is correct, thus proving that my ESRA application has generated a correct OPL program.

Looking at step 5, the reader should first note that at least 3 warehouses must be opened. The reason is that the sum of the capacities of any *2* warehouses is less than 8, which is the number of stores. With 3 open warehouses it is possible to supply all stores, for example with `uppsala`, `stockholm`, and `linkoping`, which together have a capacity of 10.

The reader should then note that the optimal cost value cannot be less than 47. This value is the cost that comes from each store choosing its *cheapest* supplying warehouse. In our solution, with the cost being 49, this is the case for all stores except for store number 3 and store number 7 (with the first store being store number 0). The only way we could get a better cost is if these stores would change to their cheapest supplying warehouses. However, for store number 3, its cheapest supplying warehouse being `stockholm`, we would have to open up a new warehouse to the dispense of 10. For store number 7, its cheapest supplying warehouse being `vasteras`, we cannot choose this warehouse because

its maximum capacity of 1 is already used, see store number 2. Alternatively we could have store number 2 change its supplier from `vasteras,` which costs 1, into another supplier, but any one of those has a cost of at least 4.

Thus we see that getting a better optimal value than 49 is impossible. The conclusion is that the solution from step 5 is the optimal solution.

Testing with the FPP Here, I test my ESRA application with the FPP problem, in the same way as, in the previous paragraph, I tested my ESRA application with the WLP problem.

For step 1, I use the ESRA model from the report [7]. There is one difference, which occurs in the declaration of the identifier `MinNbDis`. The ESRA model from the report uses the `ceil` function, like this:

```
int MinNbDis = ceil(card(Files)/DisSize);
```

The ESRA model here, which is listed further down, does not. Instead it uses the code:

```
int MinNbDis = card(Files)/DisSize+1;
```

Using this code has the same desired effect as using the `ceil` function. The reason I couldn't use the `ceil` function is that it caused an *invalid type error* in the OPL constraint solver application, which I never managed to solve.

```
int DisSize = ...;
enum Files ...;
int FileSizes[Files] = ...;
int MinNbDis = card(Files)/DisSize+1;
int MaxNbDis = card(Files);
range Diskettes 1..MaxNbDis;
var {Diskettes} UsedDiskettes;
var Files->UsedDiskettes Packing;
minimize
  card(UsedDiskettes)
subject to {
  card(UsedDiskettes) >= MinNbDis;
  max-map-weight(Packing,FileSizes,DisSize)
};
```

For step 2, the following OPL program is generated from the ESRA model by using my ESRA application. Like the OPL program generated for the WLP, this program seems to have no obvious errors.

```
int DisSize = ...;
enum Files ...;
int FileSizes[Files] = ...;
int MinNbDis = card(Files)/DisSize+1;
```

```
int MaxNbDis = card(Files);
range Diskettes 1..MaxNbDis;
var int UsedDiskettes[Diskettes] in 0..1;
var Diskettes Packing[Files];
minimize
  sum(I in Diskettes) UsedDiskettes[I]
subject to {
  sum(I in Diskettes)
    UsedDiskettes[I] >= MinNbDis;
  forall(J in Diskettes)
    UsedDiskettes[J]=1 =>
      sum(I in Files)
        (Packing[I]=J) * FileSizes[I]
           <= DisSize;
  forall(I in Files)
    UsedDiskettes [Packing[I]]=1;
};
display(I in Diskettes:
  UsedDiskettes[I]=1) <I>;
```

For step 3, I compare the OPL program generated by my ESRA application, with the OPL program generated by hand in the report [7]. Besides the `ceil` function, they are the same.

For step 4, I create the following input data to be used with the generated OPL program in the OPL constraint solver application. Like the input data for the WLP problem, this input data has been chosen such that finding the solution is not obvious. For example, putting files of sizes 2,3 and 5 into a one diskette would be the minimized solution for *that* diskette, but it would not be the minimized solution for the *overall* problem.

```
DisSize = 10;
Files =
{loveletter,wordfile,homepage,
javafile, excelsheet, initfile};
FileSizes = #[
  loveletter:7,
  wordfile:5,
  homepage:7,
  javafile:2,
  excelsheet:3,
  initfile:4
]#;
```

For step 5, the following solution is generated by the OPL constraint solver application. It is visible that the cost value is calculated correctly, and that the choice of packing files is not breaking the maximum diskette size constraint.

```
objective value: 3
variable UsedDiskettes:
1 - 0
2 - 0
3 - 0
4 - 1
5 - 1
6 - 1
variable Packing:
loveletter - 4
wordfile   - 5
homepage   - 6
javafile   - 4
excelsheet - 6
initfile   - 5
```

Looking at the input data from step 4, l can prove that the cost value from step 5 is the actual minimal cost value. As the reader can see, the sum of the sizes of the files is 28. As each diskette has a size of 10, this means that a minimum number of 3 diskettes must be used. As 3 is the solution given in step 5, this must be *the*, or *one of the*, optimal solutions.

3.13 Conclusion

The results from the tests done on the Warehouse Location problem, WLP, and the File Packing problem, FPP, in the previous section were successful. This shows that my ESRA application can translate ESRA models of the WLP and the FPP into *correctly working* OPL programs. Since these models are quite complex and contain most of the features in the ESRA language, it seems likely that my ESRA application works for most other ESRA models as well.

Chapter 4

Phase II: The Non-deterministic Compiler

4.1 Goals and Requirements

For the second phase the goal is to make the compiler from phase one non-deterministic. This means that one input program can be translated into several different output programs. The output programs are equal in their functionality but are implemented in different ways. Read more about this in section 2.5.

To ease my task, I have modified the languages used with the compiler. The input language to the compiler is now a simplified version of the ESRA language from phase 1. For example, only mapping variables are used — set variables and range variables have been removed.

The output language to the compiler has also been changed. It is now an extension of the OPL language, here called OPL+, which unlike OPL also allows set variables.

In the continuation of this chapter I will refer to the simplified version of ESRA as just ESRA. Both this language and OPL+ are described fully at the bottom of section 2.4 about ESRA and OPL.

Another new aspect that is introduced in this phase is the usage of *explanations*. For every produced translation line there can also be one or more lines of comments. This is a way of automatically generating comments in the different output models.

Finally the rule file must be modified. There should be a way to specify that an input statement can generate several output statements. One should also be able to state that some representations are dependent on other representations. Moreover, there should be a way to write explanations for every translation line. Finally, one should be able to specify substitutions. For example, if one has an expression represented by the parameter `P`, one should be able to say that `j` is to be substituted with `F[i]` in the expression `P`.

4.2 Solution and Methods

In this and the following sections I describe how I solved the requirements given in the previous section. The main problem was how to combine the different representations to generate all the models. Here is the overall description of my solution.

First, the parser breaks the input code into elements. As the reader might remember from section 2.5, there are input elements, output elements, objective elements and constraint elements, corresponding in turn to data declarations, variable declarations, objectives and constraints. Using the rules in the rule file, every element in the input language is translated into elements in the output language, i.e., a single element in the input code can produce several elements in the output code — this is called an *element set*. Moreover, an element can have several representations, i.e., it can produce a set of element sets — we call this an *element set choice,* or *choice* for short. The term *element item*, or just *item* for short, will be used in a general sense to mean an element, an element set, or an element set choice. Every type of element item has a corresponding class with the same name in my Java application.

Second, which is the central part of the solution, a tree representation is used. The generated element items from the rule file are inserted into this tree in a special way. The tree is then traversed and sets of elements are produced — every set represents one model. Read the next section to see how this tree works in more detail.

To make explanations work I added an explanation field to the element class. When the elements in the model object are processed together to create the string representation of the model, the explanations are extracted and inserted into that string representation.

Finally, in section 4.4, I describe how I modified the rule file.

4.3 The Element Tree

The element tree is the key part to generating all the models. In this section I describe how it works.

The generated element items (elements, element sets, and element set choices) from the rules are inserted into the tree. Every time an item is inserted, a new level of nodes at the bottom of the tree is created. If the item is an element or an element set, then *one* new node is created at every leaf. The new node contains the element or element set. If the item is an element set choice then *several* new nodes are created at every leaf. Each and every one of those nodes corresponds to one of the element sets in the choice.

From the procedure described above, it is clear that the element tree contains a lot of redundancy — all nodes at the same level in the tree are duplicates of each other. And worse, for every element set choice inserted into the tree, the number of duplicate nodes grows exponentially. However, the advantage of using this tree representation is that it's now easy to generate all the models. In fact,

all the paths in the tree, from the root to the leaves, represent one model. By traversing all the paths and collecting all the elements on the way, we obtain all the models.

Here follows an example to illustrate what I have just described. Let's say one has the following ESRA model:

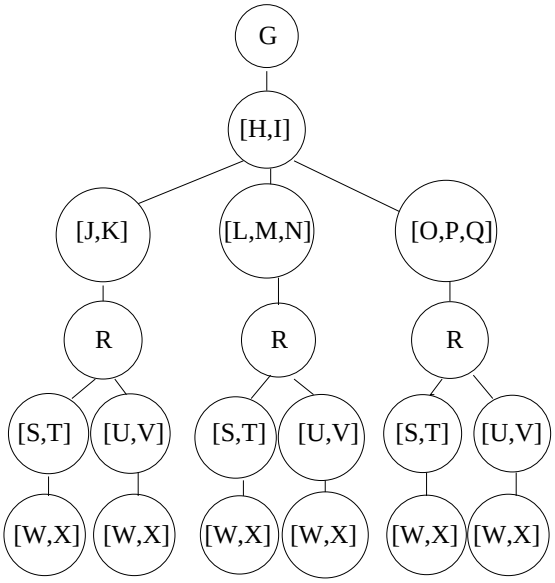
```
A;
B;
C;
D;
E;
F;
```

`A` to `F` represent ESRA statements, for example constraints, declarations, and so on. The ESRA model is then translated in the following way:

```
A  -->  G
B  -->  [H,I]
C  -->  {[J,K],[L,M,N],[O,P,Q]}
D  -->  R
E  -->  {[S,T],[U,V]}
F  -->  [W,X]
```

`G` to `X` represent elements in the output language, in our case the OPL+ language. Elements within square brackets, `[` and `]`, make up an element set. Element sets within curly braces, `{` and `}`, make up an element set choice. For example, above, `A` translates into the element `G`, while `B` translates into the element set `[H,I]`, and `C` translates into the element set choice `{[J,K],[L,M,N],[O,P,Q]}`.

Following the translation, the generated items are inserted into the tree in the same order as they were generated. The figure below shows the element tree when it is complete. Notice how the element tree branches when an element set choice is inserted into the tree. Notice also the aspect of duplicates as the element item `R` occurs three times and the element item `[W,X]` occurs six times.



Now, by traversing all the paths in the tree, we can generate all the models; the models are listed below. As models are basically element sets, I use square brackets to represent them.

```
1:  [G,H,I,J,K,R,S,T,W,X]
2:  [G,H,I,J,K,R,U,V,W,X]
3:  [G,H,I,L,M,N,R,S,T,W,X]
4:  [G,H,I,L,M,N,R,U,V,W,X]
5:  [G,H,I,O,P,Q,R,S,T,W,X]
6:  [G,H,I,O,P,Q,R,U,V,W,X]
```

In the section on non-determinism in compilation, see section 2.5, I mentioned the problem of representations depending on other representations. With the above procedure, all the combinations of representations are generated, even those so called *invalid combinations* or *invalid models* that contain representations that don't match together. The following is a description of the algorithm I used to solve this problem.

To eliminate the invalid combinations, I created special conditions for inserting an element set choice into the tree. While traversing down the path to a leaf, I checked if any of the nodes contained a variable declaration — only variable declarations can set representations. If I found such an element, I stored its representation number, a key to identify the representation. Down at the leaf, I compared the representation numbers in the choice with the representation number that I had stored. I would create nodes only for those element sets with the same representation number. In this way, paths corresponding to invalid models would never be created.

Here follows a second example illustrating an element tree. The same input

model and the same translation rules as were listed in section 2.5 have been used. To refresh the reader's memory, the program looks like this:

```
{int} V;
{int} W;
var V->W F;
solve {
  injective(F)
};
```

The illustration of the element tree that one sees below is, unlike the first example, a print-out that is generated automatically from my ESRA application. Nodes with the `parent` attribute are setting the representation and nodes with the `child` attribute are following the representation. The number after the colon after the child or the parent attribute is the representation number. Note that the algorithm on eliminating invalid models has been used: a "parent" node only has "child" nodes with the same representation number as itself below it in the tree.

```
---------------------------------------
INPUT: {int} V
INPUT: {int} W
---------------------------------------
  ---------------------------------------
  parent:1
  OUTPUT: var F[V] in W
  ---------------------------------------
    ---------------------------------------
    child:1 CONSTRAINT: alldifferent(F)
    ---------------------------------------
      ---------------------------------------
      OBJECTIVE,SOLVE
      ---------------------------------------
    ---------------------------------------
    child:1
    CONSTRAINT: forall(i in W)
                  forall(j in W)
                    i <> j =>
                      F[i] <> F[j]
    ---------------------------------------
      ---------------------------------------
      OBJECTIVE,SOLVE
      ---------------------------------------
    ---------------------------------------
    child:1
    OUTPUT: var D_F[W] in V
    CONSTRAINT: forall(i in V) forall(j in W)
```

```
                F[i] = j => D_F[j] = i
  ---------------------------------------
    ---------------------------------------
    OBJECTIVE,SOLVE
    ---------------------------------------
---------------------------------------
parent:2
OUTPUT: var F[V,W] in 0..1
CONSTRAINT: forall(j in W) sum(i in V)
               F[i,j] = 1
---------------------------------------
  ---------------------------------------
  child:2
  CONSTRAINT: forall(i in W) sum(i in V)
                F[i,j] <= 1
  ---------------------------------------
    ---------------------------------------
    OBJECTIVE,SOLVE
    ---------------------------------------
---------------------------------------
parent:3
OUTPUT: var {V} F[W];
CONSTRAINT: union all(j in W) F[j] = V
CONSTRAINT: forall(i in W) forall(j in W)
              i <> j =>
                F[i] inter F[j] = {}
---------------------------------------
  ---------------------------------------
  child:3
  CONSTRAINT: forall(i in W)
                card(F[j]) <= 1
  ---------------------------------------
    ---------------------------------------
    OBJECTIVE,SOLVE
    ---------------------------------------
```

4.4 Enhancing the Rule Converter

As mentioned in the first section of this chapter, phase 2 added new requirements to the rule file system. In this section I describe how I solved these requirements.

First, the appearance of the rule file was modified. Here is how the new rule file system lets one write rules. The examples are based on the rules for mapping variables and the injective constraint, given in section 2.5.

```
decl mappingVarDecl(F,V,W) {:
```

```
   =>    var F[V] in W;
    |    parent:1;V:intset;W:intset;
   =>    % a boolean matrix
         var F[V,W] in 0..1;
         % should be a many-to-one mapping
         forall(j in W)
           sum(i in V) F[i,j] = 1;
    |    parent:2;V:intset;W:intset;
   =>    var {V} F[W];
         union all(j in W) F[j] = V;
         forall(i in W) forall(j in W)
           i <> j => F[i] inter F[j] = {};
    |    parent:3;V:intset;W:intset
:}
cons injectiveConstraint(F) {:
   =>    alldifferent(F);
    |    child:1;F:varmap(V->W)
   =>    forall(i in W) forall(j in W)
           i <> j => F[i] <> F[j];
    |    child:1;F:varmap(V->W)
   =>    var D_F[W] in V;
         forall(i in V) forall(j in W)
           F[i] = j => D_F[j] = i;
    |    child:1;F:varmap(V->W)
   =>    forall(i in W) sum(i in V)
           F[i,j] <= 1;
    |    child:2;F:varmap(V->W)
   =>    forall(i in W) card(F[j]) <= 1;
    |    child:3;F:varmap(V->W)
:}
```

As the reader can see, the basic syntax of the rule file is still similar to that of phase 1. A block of lines starting with an arrow (=>) makes up the *output part*, and a block of lines starting with a vertical bar (|) makes up the *condition part*. Each such pair consisting of an output part and a condition part is called a *subrule*. However, there is a major difference. In phase 1, the translated result was the output from the first subrule that matched the input, i.e., the output came from only *one* subrule. Here, in phase 2, *several* subrules are allowed to match the input. The translated result will be an element set choice, in which the element sets correspond to the output parts of *all* the matching subrules. This solves the main requirement — there can now be several representations for one input statement.

Next was the problem of representations depending on other representations. I solved this by adding new attributes to the conditions. Representations with the `child` attribute depend on representations with the `parent` attribute. The two can only be combined if they have the same representation number (given

after the colon).

The new rule file also allows one to write explanations. As can be seen by the two explanations in the mapping rule, the explanations are placed above the statement they refer to. To distinguish them from normal code statements, the line or lines that make up the explanation start with a percent sign (%).

Finally, there is the issue of specifying substitutions in the rules. The example below shows how it's done. In this case the substitution takes place on line 2. `Q` is an expression in which all occurrences of `j` are substituted with `F[i]`.

```
expr sumExpression(i,j,F,Q) {:
   =>    % substitute j
         sum(i in V) Q[:j/F[i]:];
    |    child:1;F:varmap(V->W)
   =>    % sum all elements
         sum(i in V) sum(j in W)
           F[i,j] * Q;
    |    child:2;F:varmap(V->W)
   =>    var F_B[V,W] in 0..1;
         forall(i in V) forall(j in W)
           F_B[i,j] = 1 <=> i in F[j];
         sum(i in V)
           sum(j in W)
             F_B[i,j] * Q;
    |    child:3;F:varmap(V->W)
:}
```

Besides the appearance of the rule file, the new requirements have also affected the functionality of the rule converter program. Here is how the rule for sum expressions would be translated into Java (Note: I have omitted the Java code for the last two productions — the code is really three times as large).

```
public ElementSetChoice sumExpression(
  Object i_param,Object j_param,
  Object F_param,Object Q_param) throws
  UndefinedIdentifierException,
  InvalidTypeException {
ElementSetChoice choice=new ElementSetChoice();
String i; String j; String F; String Q;
{
  ElementSet es = ElementSet.createChild(1);
  i = getStringValue(i_param,1);
  addExtraElements(i_param,1,es);
  j = getStringValue(j_param,1);
  addExtraElements(j_param,1,es);
  F = getStringValue(F_param,1);
  addExtraElements(F_param,1,es);
  Q = getStringValue(Q_param,1);
```

```
  addExtraElements(Q_param,1,es);
  SymbolData FData = symbolTable.lookup(F);
  if (FData == null)
   throw new UndefinedIdentifierException(
    (SymbolInfo)F_param);
  String V = FData.mappingDomain();
  String W = FData.mappingCodomain();
  if ((true) && (FData.isMappingVariable())) {
    Explanation expl;
    expl = new Explanation();
    expl.addLine("substitute "+j+"");
    partial(es,expl,tab("sum("+i+" in "+V+
     ") \n",0) +
     tab(""+TextSubstituter.substituteId(
     ""+Q+"",""+j+"",""+F+"["+i+"]")+"",1));
     choice.addElementSet(es);
  }
}
{
  ...
  ...
  ...
}
{
  ...
  ...
  ...
}
if (choice.isEmpty())
  throw new InvalidTypeException();
return choice;
}
```

The structure of the new Java code is a lot different from that of the Java code generated in phase 1.

To begin with, the input parameters are no longer of type `SymbolInfo` but of type `Object`. This is because parameters can now also be of type `ElementSetChoice`. The method `getStringValue` is called to distinguish between the two types and retrieve the correct string value of the parameter. The method `addExtraElements` is called to retrieve the other optional elements that are associated with the string value.

Another change is that the method no longer returns an `ElementSet` object. Since rules can generate several representations, the method now returns an `ElementSetChoice` object. Further on, `Explanation` objects are created and added to their corresponding element objects.

4.5 Testing and Results

In this section I test that my ESRA application works correctly. As the reader knows, the ESRA application takes an input program in the ESRA language and produces several output programs in the OPL+ language. As the OPL+ language is not compilable, I cannot check that the output programs themselves work correctly (compare with steps 4, 5 and 6 from the test procedure used in phase 1, see section 3.12). I can only check that the ESRA application produces the right output programs, based on the rules in the rule file.

The rule file consists of five rules. Three of these rules are the rules for the mapping variable declaration, the injective constraint, and the sum expression, which were described in the previous section. The two other rules are the rules for the forall constraint and the inverse image of a mapping. Each of the five rules gets listed further down in the section in the moment it is applied for the first time.

To test the ESRA application I will accomplish the following tasks:

- Task 1: create an ESRA model that makes use of many of the features implemented in the ESRA application.
- Task 2: using the rules in the rule file, produce the output programs *by hand — not by using the compiler.*
- Task 3: run the ESRA model thru the ESRA application to generate the output programs.
- Task 4: compare the compiler-generated output programs with the output programs produced by hand.

The objective of these tasks is to test that the ESRA application generates the correct output programs for the ESRA model. For a successful outcome of the test, the output programs produced by hand in task 2 should be equal to the output programs generated by the compiler in task 3. With the ESRA model being complex, such a successful outcome should mean that the ESRA application will work correctly for many other ESRA models as well.

Task 1, creating the ESRA model For step 1, I create the following ESRA model. It consists of two integer sets `V` and `W`; one mapping variable `F` from `V` into `W`; an injective constraint on `F`; and a complex `forall` constraint.

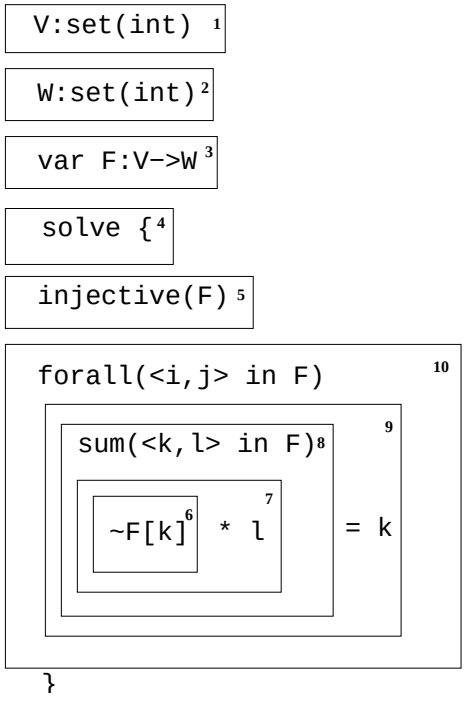
```
{int} V;
{int} W;
var V->W F;
solve {
  injective(F);
  forall(<i,j> in F)
    sum (<k,l> in F)
```

```
        ~F[k] * l = k
};
```

This ESRA model is small and does not represent anything meaningful. However, despite its size, it actually makes use of all five rules in the rule file. Also, three of the rules, the rules for the forall constraint, the sum expression, and the inverse image of a mapping, are used in a nested way.

Task 2, producing the output programs by hand In this extensive paragraph I will perform step by step the creation of the the output programs of the ESRA model *by hand*. The purpose is to check that the generated output programs from the compiler are the correct ones.

I start by dividing the ESRA model into numbered parts, 1 to 10, as the figure below shows.



For each part, element items in the OPL+ language are produced. By using the rules in the rule file, listed in the beginning of this section, I can manually determine these element items.

When illustrating the element items in the continuation of this section, I use semi-colons to separate elements from each other in an element set. Each element set in an element set choice is then prefixed with the number of the representation it applies to followed by a colon. If several element sets apply to the same representation, these are called *alternatives*. An element set that is an alternative of a representation is prefixed with the representation number

followed by a colon, a comma sign (,) and its *alternative number*. The alternative number pertains to the place the element set has among the other element sets that apply to the same representation; the first element set has alternative number 1, the second element set alternative number 2, and so on. For example, in the element set choice illustration below, the first element set applies to representation 1, the second and the third element set to representation 2, and the fourth element set to representation 3. The second and the third element set are distinguished from each other by their alternative numbers, 1 and 2, respectively.

```
1:   F[i]=F[j];
     i>=j;
2,1: F[i,j]=0;
2,2: F[i,j]=1;
     i<=j => i=0;
3:   F[i]=0;
```

Parts 1 and 2 These parts are the integer set declarations `{int} V` and `{int} W`. There exists no rule in the rule file for translating these. Instead, the translation scheme is coded directly into the semantic code of the parser specification file (`esra_pass2.cup`). In this file, it is specified that integer set declarations should be translated into themselves:

```
{int} V;
{int} W;
```

Part 3 This part is the mapping variable declaration `var V->W F`. I use the rule for mapping variable declarations from the rule file:

```
decl mappingVarDecl(F,V,W) {:
   =>    var F[V] in W;
    |    parent:1;V:intset;W:intset;
   =>    var F[V,W] in 0..1;
         forall(j in W)
           sum(i in V) F[i,j] = 1;
    |    parent:2;V:intset;W:intset;
   =>    var {V} F[W];
         union all(j in W) F[j] = V;
         forall(i in W) forall(j in W)
           i <> j => F[i] inter F[j] = {};
    |    parent:3;V:intset;W:intset
:}
```

This should produce the following element set choice:

```
1: var F[V] in W;
2: var F[V,W] in 0..1;
```

```
   forall(j in W)
     sum(i in V) F[i,j] = 1;
3: var {V} F[W];
   union all(j in W) F[j] = V;
   forall(i in W) forall(j in W)
     i <> j => F[i] inter F[j] = {};
```

Part 4 This part, `solve`, is the objective of the model. As is listed in the parser specification file, it translates into an element of itself.

Part 5 This part is the injective constraint, `injective(F)`. To translate it, I use the rule for the injective constraint in the rule file:

```
cons injectiveConstraint(F) {:
   =>    alldifferent(F);
    |    child:1;F:varmap(V->W)
   =>    forall(i in W) forall(j in W)
           i <> j => F[i] <> F[j];
    |    child:1;F:varmap(V->W)
   =>    var D_F[W] in V;
         forall(i in V) forall(j in W)
           F[i] = j => D_F[j] = i;
    |    child:1;F:varmap(V->W)
   =>    forall(i in W) sum(i in V)
           F[i,j] <= 1;
    |    child:2;F:varmap(V->W)
   =>    forall(i in W) card(F[j]) <= 1;
    |    child:3;F:varmap(V->W)
:}
```

It produces the following element set choice:

```
1,1: alldifferent(F);
1,2: forall(i in W) forall(j in W)
     i <> j => F[i] <> F[j];
1,3: var D_F[W] in V;
     forall(i in V) forall(j in W)
       F[i] = j => D_F[j] = i;
2:   forall(i in W) sum(i in V)
       F[i,j] <= 1;
3:   forall(i in W) card(F[j]) <= 1;
```

Parts 6 to 10 These parts belong to the `forall` constraint and are all nested within each other. The structure of the parts corresponds to the parse tree of the ESRA model. The most inner part is the part that is parsed and translated first by the compiler. Its result is passed up to the next innermost part, and

so on. The complete translation of the `forall` constraint is the result from the outermost part, part 10.

Part 6 This part is the inverse image expression `~F[k]`. Listed below is the rule for inverse image expressions in the rule file, which I use for translating the expression.

```
expr mappingImageExpression(F,i) {:
   =>    F[i];
    |    child:1;F:varmap(V->W)
   =>    sum(j in W) F[i,j] * j;
    |    child:2;F:varmap(V->W)
   =>    var F_B[V,W] in 0..1;
         forall(i in V) forall(j in W)
           F_B[i,j] = 1 <=> i in F[j];
         sum(j in W) F_B[i,j] * j;
    |    child:3;F:varmap(V->W)
:}
```

By applying the rule, the following element set choice should be produced. Note that I have substituted all occurrences of `i` with `k`.

```
1: F[k];
2: sum(j in W) F[k,j] * j;
3: var F_B[V,W] in 0..1;
   forall(k in V) forall(j in W)
     F_B[k,j] = 1 <=> k in F[j];
   sum(j in W) F_B[k,j] * j;
```

Part 7 This part is the arithmetic expression `~F[k]*1`. As is specified in the parser specification file, all arithmetic expressions should translate into themselves. Thus, I should take the translated result of `~F[k]` and combine it with the code string "`* 1`". Because the result of `~F[k]` is an element set choice, (the element set choice from part 6), the procedure is complex. The code string should be added to *all* element sets in the choice. For each element set, it should be appended to the *partial element*. The partial element is a special type of element that is not represented by any of the four standard elements, i.e., input elements, output elements, constraint elements and objective elements. The partial elements are used to represent subparts of other elements. For example, in the third element set listed above, the sum expression on the last line is a partial element. This expression is *not* one of the four standard elements, but it is used to eventually create the forall constraint of part 10, which *is* one of the four standard elements.

Applying the procedure, appending "`* 1`" to all partial elements in the element set choice from part 6, I get this new element set choice:

```
1: F[k] * l;
2: sum(j in W) F[k,j] * j * l;
3: var F_B[V,W] in 0..1;
   forall(k in V) forall(j in W)
     F_B[k,j] = 1 <=> k in F[j];
   sum(j in W) F_B[k,j] * j * l;
```

Part 8 This part is the sum expression used in the ESRA model:

```
sum(<k,l> in F)
  ~F[k] * l
```

For translating this, I use the rule for sum expressions in the rule file:

```
expr sumExpression(i,j,F,Q) {:
   =>    sum(i in V) Q[:j/F[i]:];
    |    child:1;F:varmap(V->W)
   =>    sum(i in V) sum(j in W)
           F[i,j] * Q;
    |    child:2;F:varmap(V->W)
   =>    var F_B[V,W] in 0..1;
         forall(i in V) forall(j in W)
           F_B[i,j] = 1 <=> i in F[j];
         sum(i in V)
           sum(j in W)
             F_B[i,j] * Q;
    |    child:3;F:varmap(V->W)
:}
```

Notice the parameters in the signature of the rule. For our sum expression, `i` corresponds to `k`, `j` corresponds to `l`, and `Q` corresponds to `~F[k] * l`. As the result of `~F[k] * l` is an element set choice (the element set choice from part 8), the procedure is again complex. For each subrule R, an element set S is produced by the rule. In R, every occurrence of the string "`Q`" should be replaced. The element set E in `Q`, that applies to the same representation as does R, should be used for the replacement. Its partial element, X, should be the new value. The remaining elements in E are added to S.

Applying the procedure on the rule for sum expressions, I obtain the element set choice listed below. All occurrences of `i` and `j` have been replaced by the identifiers `k` and `l`, and all occurrences of "`Q`" have been replaced by the correct partial elements. Note that the substitution expression `Q[:j/F[i]:]` from the first subrule comes into effect. With `i` and `j` substituted, the substitution expression is really `Q[:l/F[k]:]`. In the partial element that corresponds to this occurrence of `Q`, all occurrences of `l` should be substituted with `F[k]`. With the partial element being the arithmetic expression `F[k]*l` from the first element set for part 7, the result of the substitution is `F[k]*F[k]`.

```
1: sum(k in V)
     F[k] * F[k];
2: sum(k in V) sum(j in W)
     F[k,l] * sum(j in W) F[k,j] * j * l;
3: var F_B[V,W] in 0..1;
   var F_B[V,W] in 0..1;
   forall(k in V) forall(j in W)
     F_B[k,l] = 1 <=> k in F[l];
   forall(k in V) forall(j in W)
     F_B[k,j] = 1 <=> k in F[j];
   sum(k in V)
     sum(l in W)
       F_B[k,l] * sum(j in W) F_B[k,j] * j * l;
```

The observant reader might notice that the third element set has a strange feature, which could be interpreted as erroneous. It contains two exact instances of the same variable declaration:

```
var F_B[V,W] in 0..1;
var F_B[V,W] in 0..1;
```

This strange feature is dealt with in the end of the section.

Part 9 This part is the equality relation

```
sum(<k,l> in F)
  ~F[k] * l = k
```

Its left operand is the sum expression; its right operand is the identifier `k`. Similar to part 7, the result of the left operand is an element set choice (the element set choice from part 8). The translation is done by appending the code string "`= k`" to all the partial elements in the choice. By doing this, I get this new element set choice:

```
1: sum(k in V)
     F[k] * F[k] = k;
2: sum(k in V) sum(j in W)
     F[k,l] * sum(j in W)
       F[k,j] * j * l = k;
3: var F_B[V,W] in 0..1;
   var F_B[V,W] in 0..1;
   forall(k in V) forall(j in W)
     F_B[k,l] = 1 <=> k in F[l];
   forall(k in V) forall(j in W)
     F_B[k,j] = 1 <=> k in F[j];
   sum(k in V)
     sum(l in W)
       F_B[k,l] * sum(j in W)
         F_B[k,j] * j * l = k;
```

Final part, part 10 This part is the forall constraint

```
forall(<i,j> in F)
  sum(<k,l> in F)
    ~F[k] * l = k
```

The rule for the forall constraint in the rule file is used to do the translation:

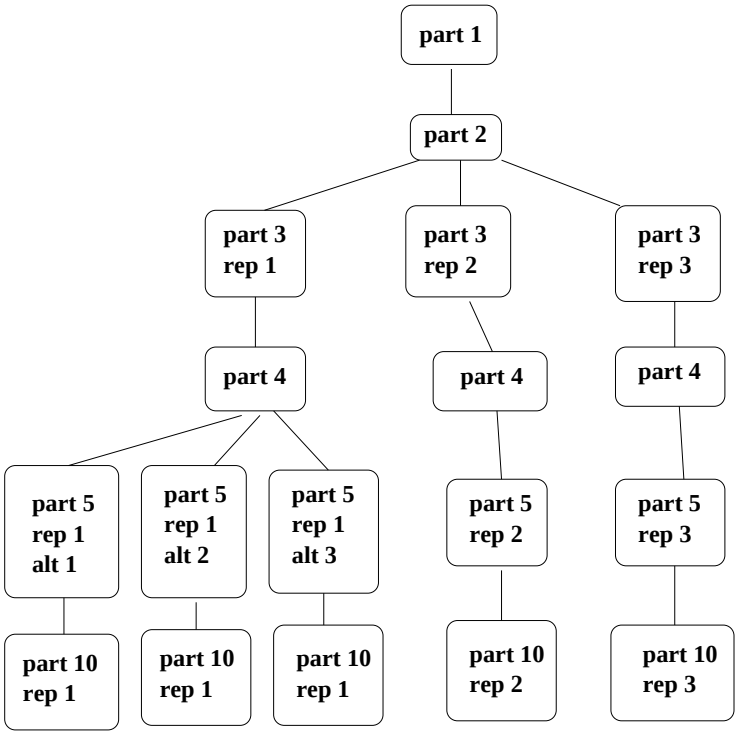
```
cons forallConstraint(i,j,F,P) {:
   =>    forall(i in V) P[:j/F[i]:];
    |    child:1;F:varmap(V->W)
   =>    forall(i in V) forall(j in W)
           F[i,j] = 1 => P;
    |    child:2;F:varmap(V->W)
   =>    forall(i in V) forall(j in W)
           i in F[j] => P;
    |    child:3;F:varmap(V->W)
:}
```

Like for part 8, one of the parameters to the rule is represented by an element set choice; this is the parameter `P`, which is represented by the element set choice from part 9. I use the procedure that was described for part 8, and I obtain a new element set choice, which is listed below. Unlike for part 8, the substitution expression `P[:j/F[i]:]` in the first subrule does *not* come into effect. This is because the relation `sum(<k,l> in F) ~F[k] * l = k`, which is the partial element in the first element set from part 9, doesn't contain any occurrences of the identifiers that are represented by the parameters `i` and `j` in the rule (the identifiers in this case happen to have the same names as the parameters: `i` and `j`).

```
1: forall(i in V) sum(k in V)
     F[k] * F[k] = k;
2: forall(i in V) forall(j in W)
     F[i,j] = 1 =>
       sum(k in V) sum(j in W)
         F[k,l] * sum(j in W)
           F[k,j] * j * l = k;
3: var F_B[V,W] in 0..1;
   var F_B[V,W] in 0..1;
   forall(k in V) forall(j in W)
     F_B[k,l] = 1 <=> k in F[l];
   forall(k in V) forall(j in W)
     F_B[k,j] = 1 <=> k in F[j];
   forall(i in V) forall(j in W)
     i in F[j] => sum(k in V)
       sum(l in W)
         F_B[k,l] * sum(j in W)
           F_B[k,j] * j * l = k;
```

Creating the element tree As all the element items have been determined, I can now create the element tree. The element items that were obtained from translating parts 1, 2, 3, 4, 5, and 10 of the ESRA model are the items that I insert into the tree. When inserting the items, I make sure that I do not create branches that give rise to invalid models, i.e., models that contain items with conflicting representations.

Below, a figure of the tree is shown when it is complete. In the figure, the elements or element sets that exist in the nodes are identified by one or more of the following: (1) the number of the part of the ESRA model that they were produced from, (2) optionally the number of the representation that they apply to, and (3) optionally the number of the alternative of that representation.



Creating the models I can now create all the models by traversing the paths in the element tree. All of the five models that are created will be treated in their own paragraphs below. For each paragraph, I will simultaneously perform tasks 3 and 4, so that I can verify that the ESRA application has generated the correct model.

Model 1 By traversing the left-most path in the element tree, I obtain the element items that belong to model 1:

```
(part 1)
  V{int} Vset(int);
(part 2)
  {int} W;
(part 3,rep 1)
  var F[V] in W;
(part 4)
  solve
(part 5,rep 1,alt 1)
  alldifferent(F);
(part 10,rep 1)
  forall(i in V) sum(k in V)
    F[k] * F[k] = k;
```

The following is an excerpt of the print-out generated from the ESRA application, representing model 1.

```
Model 1:
---------------------------------------
{int} V;
{int} W;
var F[V] in W;
solve {
  alldifferent(F);
  forall(i in V) sum(k in V)
      F[k] * F[k] = k
};
```

A comparison of the two models shows that they contain the same element items. The two models are therefore equal. The fact that the element items might not be ordered in the same way is irrelevant.

Model 2 By traversing the second path in the element tree, I obtain the element items that belong to model 2:

```
(part 1)
  {int} V;
(part 2)
  {int} W;
(part 3,rep 1)
  var F[V] in W;
(part 4)
  solve
(part 5,rep 1,alt 2)
  forall(i in W) forall(j in W)
    i <> j => F[i] <> F[j];
(part 10,rep 1)
```

```
  forall(i in V) sum(k in V)
    F[k] * F[k] = k;
```

Below is the compiler-generated excerpt representing model 2.

```
Model 2:
----------------------------------------
{int} V;
{int} W;
var F[V] in W;
solve {
  forall(i in W) forall(j in W)
    i <> j => F[i] <> F[j];
  forall(i in V) sum(k in V)
    F[k] * F[k] = k
};
```

A comparison shows that the two models are equal.

Model 3 Path number 3 in the element tree yields the element items that belong to model 3:

```
(part 1)
  {int} V;
(part 2)
  {int} W;
(part 3,rep 1)
  var F[V] in W;
(part 4)
  solve
(part 5,rep 1,alt 2)
  var D_F[W] in V;
  forall(i in V) forall(j in W)
    F[i] = j => D_F[j] = i;
(part 10,rep 1)
  forall(i in V) sum(k in V)
    F[k] * F[k] = k;
```

Below is the excerpt of model 3 generated by the compiler.

```
Model 3:
----------------------------------------
{int} V;
{int} W;
var F[V] in W;
var D_F[W] in V;
solve {
```

```
  forall(i in V) forall(j in W)
    F[i] = j => D_F[j] = i;
  forall(i in V) sum(k in V)
    F[k] * F[k] = k
}
```

A comparison of the two models shows that model 3 has been generated correctly by the ESRA application.

Model 4 Path number 4 in the element tree yields the element items that belong to model 4:

```
(part 1)
  {int} V;
(part 2)
  {int} W;
(part 3,rep 2)
  var F[V,W] in 0..1;
  forall(j in W)
    sum(i in V) F[i,j] = 1;
(part 4)
  solve
(part 5,rep 2)
  forall(i in W) sum(i in V)
    F[i,j] <= 1;
(part 10,rep 2)
  forall(i in V) forall(j in W)
    F[i,j] = 1 =>
      sum(k in V) sum(l in W)
        F[k,l] * sum(j in W)
          F[k,j] * j * l = k;
```

Below is the excerpt of model 4 generated by the compiler.

```
Model 4:
----------------------------------------
{int} V;
{int} W;
var F[V,W] in 0..1;
solve {
  forall(j in W) sum(i in V)
    F[i,j] = 1;
  forall(i in W) sum(i in V)
    F[i,j] <= 1;
  forall(i in V) forall(j in W)
    F[i,j] = 1 => sum(k in V)
      sum(l in W)
```

```
        F[k,l] * sum(j in W)
          F[k,j] * j * l = k
};
```

A comparison of the two models shows that model 4 has been generated correctly by the ESRA application.

Model 5 Path number 5 in the element tree yields the element items that belong to model 5:

```
(part 1)
  {int} V;
(part 2)
  {int} W;
(part 3,rep 3)
  var {V} F[W];
  union all(j in W) F[j] = V;
  forall(i in W) forall(j in W)
    i <> j => F[i] inter F[j] = {};
(part 4)
  solve
(part 5,rep 3)
  forall(i in W) card(F[j]) <= 1;
(part 10,rep 3)
  var F_B[V,W] in 0..1;
  var F_B[V,W] in 0..1;
  forall(k in V) forall(j in W)
    F_B[k,l] = 1 <=> k in F[l];
  forall(k in V) forall(j in W)
    F_B[k,j] = 1 <=> k in F[j];
  forall(i in V) forall(j in W)
    i in F[j] => sum(k in V)
      sum(l in W)
        F_B[k,l] * sum(j in W)
          F_B[k,j] * j * l = k;
```

Below is the excerpt of model 5 generated by the compiler.

```
Model 5:
----------------------------------------
{int} V;
{int} W;
var {V} F[W];
var F_B[V,W] in 0..1;
var F_B[V,W] in 0..1;
```

```
solve {
  union all(j in W) F[j] = V;
  forall(i in W) forall(j in W)
    i <> j => F[i] inter F[j] = {};
  forall(i in W)
    card(F[j]) <= 1;
  forall(k in V) forall(j in W)
    F_B[k,j] = 1 <=> k in F[j];
  forall(k in V) forall(l in W)
    F_B[k,l] = 1 <=> k in F[l];
  forall(i in V) forall(j in W)
    i in F[j] => sum(k in V)
      sum(l in W)
        F_B[k,l] * sum(j in W)
          F_B[k,j] * j * l = k
};
```

A comparison of the two models shows that model 5 has been generated correctly by the ESRA application.

As all output models generated by hand are equal to the output models generated by the compiler, it means that the ESRA application has successfully compiled the ESRA model.

The issue of duplicates As the reader might have noticed, model 5 has a strange feature, which could be interpreted as an error. In this model there are two identical declarations of the array variable `F_B`:

```
var F_B[V,W] in 0..1;
var F_B[V,W] in 0..1;
```

There are also two constraints that, although not identical, are semantically equal:

```
forall(k in V) forall(j in W)
  F_B[k,j] = 1 <=> k in F[j];
forall(k in V) forall(l in W)
  F_B[k,l] = 1 <=> k in F[l];
```

The reason is that there are two rules that produce the same declaration and the same constraint. They are the third subrule of the rule for the sum expression

```
=>   var F_B[V,W] in 0..1;
     forall(i in V) forall(j in W)
       F_B[i,j] = 1 <=> i in F[j];
     sum(i in V)
       sum(j in W)
         F_B[i,j] * Q;
 |   child:3;F:varmap(V->W)
```

and the third subrule of the rule for inverse image expressions

```
=>   var F_B[V,W] in 0..1;
     forall(i in V) forall(j in W)
       F_B[i,j] = 1 <=> i in F[j];
     sum(j in W) F_B[i,j] * j;
 |   child:3;F:varmap(V->W)
```

Their purpose is to convert the array `F`, used for the third representation, into a new matrix `F_B`, used for the second representation. In this way, the third subrule can use the translation technique of the second subrule.

A simple solution to this problem would be to create different names for the two matrices, for example one called `F_B1` and one called `F_B2`. However, the problem occurs in other situations as well. For example, a program containing two inverse image expressions, such as `~F[j]` and `~F[k]` also produces two identical declarations of the matrix `F_B`. Here, one cannot solve the problem by using different names, because the declarations are both produced by the same rule.

Despite this problem I will not spend time solving it. The reason is that it has been agreed on that my targeted ESRA application is not required to handle this issue.

The issue of multiple mapping variables There is also another issue, although not visible from the test, that can be interpreted as erroneous. This concerns the usage of *several* mapping variables in the ESRA model. The non-deterministic compiler was designed in mind of only being capable of dealing with *one* mapping variable. When using *several* mapping variables in the ESRA model, this presents a problem to the compiler. To understand this problem, look at the following ESRA model that uses two mapping variables:

```
{int} V;
{int} W;
var V->W F;
var V->W G;
solve {
  injective(F);
  injective(G)
};
```

When translating the constraint `injective(F)`, the translated result will contain several occurrences of the mapping variable `F`. For each model, it is important that the representation of `F` used here, is the same as the representation used in the declaration of `F`. In section 4.3, the procedure for inserting element items into the element tree was described. The procedure is to traverse down the tree and add the new element item to all the leaves. For element set choices, only element sets that have the same representation number as the *parent* in the branch are added to the leaf. The parent is an element item that has been

produced by a rule in the rule file with the `parent` attribute. With the current rule file, mapping variable declarations are parents. When a program has two mapping variables, there will be *two* parents in the branch. The element set choice to be inserted checks the representation set by the *closest* parent in the branch, *believing there is only one parent*. In the case of `injective(F)`, the closest parent is the declaration of `G`. The translation of `injective(F)` therefore uses the representation of `G`, when instead it should have used the representation of `F`.

Listed below is an excerpt of the element tree that is generated by the ESRA application, when run on the ESRA model above. The excerpt shows one of the branches in the tree, which represents one of the output programs. The element items in the branch have been numbered from 1 to 6.

```
-------------------------------------- (1)
INPUT: {int} V
INPUT: {int} W
--------------------------------------
-------------------------------------- (2)
parent:1
OUTPUT: var F[V] in W
--------------------------------------
-------------------------------------- (3)
parent:2
OUTPUT: var G[V,W] in 0..1
CONSTRAINT: forall(j in W) sum(i in V)
              G[i,j] = 1
--------------------------------------
-------------------------------------- (4)
child:2
CONSTRAINT: forall(i in W) sum(i in V)
              F[i,j] <= 1
--------------------------------------
-------------------------------------- (5)
child:2
CONSTRAINT: forall(i in W) sum(i in V)
              G[i,j] <= 1
--------------------------------------
-------------------------------------- (6)
OBJECTIVE,SOLVE
--------------------------------------
```

In element item 2, `F` is declared using its representation number 1, `F`, while in element item 3, `G` is declared using its representation number 2. The problem occurs for element item 4, which is the translation of `injective(F)`. Instead of using representation 1, `F`, which is the correct representation, it uses representation 2, `F`. The generated output program will incorrectly contain occurrences of both `F` and `F`.

Like the issue concerning duplicates, I will not spend any time solving this problem. It has been agreed on that my targeted ESRA application is not required to handle ESRA models with multiple mapping variable declarations.

4.6 Conclusion

The tests performed in the previous section proved that the ESRA application could correctly generate all of the output programs for the ESRA model that was listed there. As this ESRA model was rather complex, making use of all the rules in the rule file, it seems likely that the ESRA application will work for most other ESRA models, as well.

Chapter 5

Conclusion

This work consisted of two phases which are now completed — my project has come to its conclusion.

Summary In this project I have created two different compilers. In phase 1, I created a compiler that translates models in the ESRA language into OPL programs. The ESRA language used here supports a wide range of primitives. In phase 2, I created a compiler that translates models in the ESRA language into a set of different output programs in OPL+. The ESRA language used here doesn't support as many primitives, but the choice of different output programs provides for better execution times.

For creating the compiler I used the compiler compiler help tools called JLex and JavaCUP. They let me specify the procedure for lexing and parsing in separate specification files. These files could then automatically be converted to working Java programs.

Work was spent on analyzing and investigating how JLex and JavaCUP worked in detail. This led to the solutions of how the JLex generated tokenizer could be connected with the JavaCUP generated parser, how to create a two-pass compiler with JavaCUP, and how to report errors with JLex and JavaCUP.

There was also a wish to have the possibility of viewing the different stages of the compilation. For this I created two programs called the token list generator generator and the parse tree generator generator. These programs altered the behavior of the JLex and JavaCUP specification files, so that they produced nice print-outs of the token list and the parse tree respectively.

To simplify the procedure of writing translation rules for the compiler, I created a rule converter program. The program lets one specify the rules in a simple format, and then converts these into Java methods.

For making the compiler easy to work with, I created a graphical application. This application lets one open and save files and use the menus to invoke actions in the compiler. Also, it includes a nice error handling system, which is capable of presenting informative error messages and high-lighting erroneous parts of the program.

Evaluation The main goal of the project was to implement the ESRA language. As the results of the tests showed, both for phase 1 and phase 2, this goal has been comfortably fulfilled. The compiler from phase 1 successfully translates ESRA models of two classic constraint problems into OPL programs; the OPL programs work correctly on given input data. The compiler from phase 2 successfully produces OPL+ models from a rather complex ESRA model; the models produced have been proven to be correct ones, based on the translation rules used.

Another requirement was that of flexibility; the grammar and the translation rules that the compiler uses should be easy to modify. By using JLex and JavaCUP, the grammar is defined in its own specification file. Here, the format of the grammar is very similar to that used in ASTRA reports, and thus the grammar is easy to modify. Likewise, having created a rule converter program, the translation rules are also defined in their own file, the rule file. As the format for creating the rules is very user-friendly, the translation rules are easy to modify.

Lessons learned There are some things that I wish I would have done differently in the project.

First, it is the matter of how I created the three programs: token list generator generator, parse tree generator generator, and the rule converter. Each of these programs had the task of parsing an input file: the token list generator generator parsed the JLex specification file, the parse tree generator generator parsed the JavaCUP specification file, and the rule converter parsed the rule file. As Perl is a good programming language for text processing, I used Perl to create these three programs. The main principle used in each program is that of *cut and paste* using Perl's pattern matching and substitution mechanisms. Using this principle made it hard to update the code as the program grew in size. For example, modifying the rule converter program to deal with the new changes to the rule file in phase 2 was almost impossible. An alternative approach to creating the three programs would have been to, ingeniously, use JLex and JavaCUP (!). JLex and JavaCUP are designed to deal with text that has a well-defined grammatical structure, as is the case with the JLex specification file, the JavaCUP specification file, and the rule file.

Second, it is the matter of the method used by the non-deterministic compiler for generating all the output programs belonging to a certain ESRA model. With the current method, as is explained in detail in section 4.3, the compiler constructs a tree of translated items by appending every item that is inserted to all the leaves in the tree. This scheme makes it easy to generate all the models (each model is represented by a path in the tree) but wastes a lot of storage, as each item is represented several times in the tree. A better approach would have been to create a *linked list* of the items, and then generate all the models, by using an algorithm that is capable of iterating over all the combinations of the items' representations.

Third and final, it is a matter concerning the basic principle used in the

compilation process. When creating a compiler, two different techniques are commonly used regarding the parsing and the translation. With the first technique, the parsing and the translation are done in the same step: the input program is broken apart and reconstructed at the same time; the translation is said to be done *on the fly*. With the second technique, the parsing and the translation are done separately. An intermediate step is used, in which a concrete parse tree object is created. After that, the translation can be performed by processing this parse tree object. The parse tree object can be a tree of linked Java objects, each object representing a part of the input program, and each of its attributes representing subparts of that part.

In my compiler the first of the mentioned techniques is used. I soon regretted this, because there were several points in the translation where I needed access to information concerning other parts of the parse tree. This occurred, for example, when processing variable declarations, which are specified like this in the grammar:

```
VarDecl ::= VAR TypeVar:typeVar ID:name
```

To create the translation for the `VarDecl` statement, I need information about the `TypeVar` statement which is located below the `VarDecl` statement in the tree. As the translation is done *on the fly*, the information about the `TypeVar` statement is lost when the translation process reaches the `VarDecl` statement. Up till now, I have solved this problem by creating Java objects for every non-terminal that I need to save information about. The information is stored in the attributes of the objects. This is in a nut shell how the second technique works. Had I incorporated the second technique from the beginning, the scheme of using Java objects to access information about the non-terminals in the parse tree would *already be a natural part of the compiler*.

Future work Even though my work here is done, the *project* is by no means a finished chapter. Here is a list of some of the things that can be done later to continue the project:

- The problem with duplicates, from phase 2, that was described in the end of section 4.5, *Testing and Results,* can be solved, possibly by modifying the rule converter program.
- The compiler can be updated to handle declarations of *multiple* mapping variables. The way to achieve this would possibly be to modify the implementation of the element tree.
- For the non-deterministic compiler, the OPL+ language can be replaced by another output language *that is compilable*. In this way, it would be possible to test the execution times of the different output programs.
- As a final fantasy, which would mean very ambitious work, but certainly is possible, the specification of the grammar from the JavaCUP specification

file and the specification of the translation rules from the rule file could be joined together to form a new type of specification file. A fictitious example of this is listed below.

```
Expr -> TILDE Id:F LBRACK Deref:i RBRACK {:
  => F[i];
   | child:1;F:varmap(V->W)
  => sum(j in W) F[i,j] * j;
   | child:2;F:varmap(V->W)
  => var F[V,W] in 0..1;
     forall(i in V) forall(j in W)
       F_B[i,j] = 1 <=> i in F[j];
     sum(j in W) F_B[i,j] * j;
   | child:3;F:varmap(V->W)
:}
Expr -> SUM LPAREN LESS Id:i COMMA Id:j GREATER
        IN Id:F RPAREN Expr:Q {:
   => sum(i in V) Q[:j/F[i]:];
    | child:1;F:varmap(V->W)
   => sum(i in V) sum(j in W)
        F[i,j] * Q;
    | child:2;F:varmap(V->W)
   => var F_B[V,W] in 0..1;
      forall(i in V) forall(j in W)
        F_B[i,j] = 1 <=> i in F[j];
         sum(i in V) sum(j in W)
           F_B[i,j] * Q;
    | child:3;F:varmap(V->W)
:}
```

Here, the translation rule for a grammar production has been inserted into the semantic code of the grammar production. Each such grammar production combined with its translation rule is called a *grammar-translation rule*. The above shows the grammar-translation rules for the inverse image mapping expression and the sum expression. The identifiers of the labeled non-terminals in the grammar production correspond to the parameters of the rule (see section 3.10), i.e., those entities whose values will replace the occurrences of the entities in the body of the rule.

The advantage of using this scheme of directly combining the grammar production with its translation rule, compared to the current way of separating the grammar and the translation rules, is that of efficiency and abstraction. Having the grammar and the corresponding translation rules in the same location should make it easier to update the compiler. Also, the new specification file gives a higher level of abstraction, as one doesn't have to use calls to Java methods to specify the translation rules.

Appendix A

ESRA Application User's Manual

A.1 Introduction

This manual pertains to the second version of the ESRA application, i.e., that of phase 2, and is for anyone who wants to install and run the ESRA application and learn how to use it. A diskette containing the ESRA application in question is available together with the report.

A.2 How to Install

This ESRA application can only be run on UNIX platforms.

To install, load the diskette and copy the `esra2.tar.gz`[1] file to an existing directory in your file system.

To decompress the `esra2.tar.gz` file, stand in the directory and type the following command:

```
tar -zxvf esra2.tar.gz
```

This will create a directory named ESRA2.

Check for a file in your home directory called `.bashrc`. If you don't have one, create an empty one. Put the following two lines in the `.bashrc` file:

```
export ESRA_PATH=<yourpath>/ESRA2
source $ESRA_PATH/aliases
```

Exchange `<yourpath>` with the path to the directory which contains the ESRA2 directory. You should use the full path of this directory, i.e., a path starting with

[1] The reason the file is called `esra2.tar.gz` (and not simply `esra.tar.gz`) is that it refers to the ESRA application *from phase 2*.

the root symbol (/), and not a path containing for example a tilde character (~). Note also that it is very important that you don't use any extra spaces in the two lines.

For example, let's say I copy the `esra2.tar.gz` file to my directory `/home/siwr9625/EXJOBB`. The two lines that I put in my .bashrc file will be:

```
export ESRA_PATH=/home/siwr9625/EXJOBB/ESRA2
source $ESRA_PATH/aliases
```

Last, you should type the following at the command line:

```
source ~/.bashrc
```

A.3 How to Run

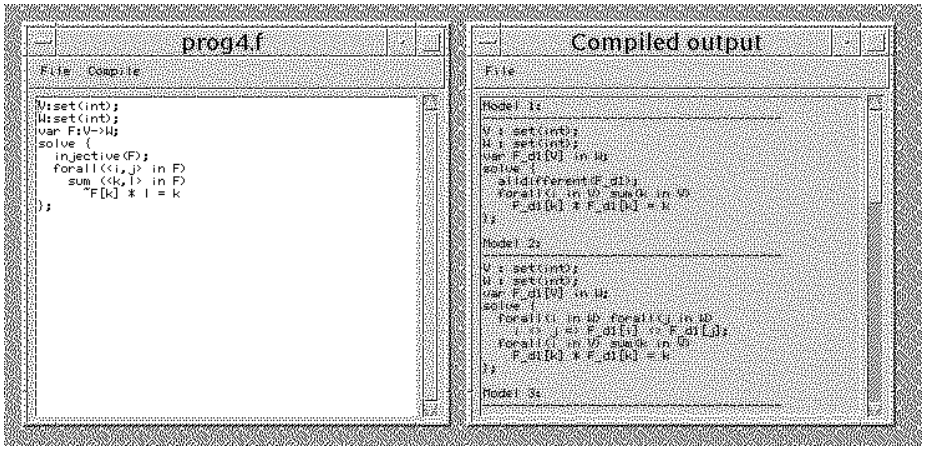
Make sure that you are running the unix shell *bash*. To find out which shell you are running, type `echo $SHELL` at the command line. If you are not running bash, simply type `bash` on the command line.

Then, stand in any directory and type `esra` to start the ESRA application. This should open the left window of the two windows that are shown below.

If this doesn't happen try typing `source ~/.bashrc` at the command line, and then try typing `esra` again

A.4 Basics

When one starts the ESRA application a window shows up. It is in this window that one writes one's ESRA programs. To compile your ESRA program, `All Models` from the `Compile` menu should be chosen. This opens up a second window that shows the result of the compilation. The `Compile` menu also has other options, and these are described in the next section. The following is a screen dump of the ESRA application with both windows opened.



A.5 Menu Options

The main window has two menus: the `File` menu and the `Compile` menu. The `File` menu is the left one of the two menus in the user interface. It is similar to a `File` menu of any other normal application. The `Compile` menu has commands associated with the compilation process. `Tokenize` shows what the list of tokens looks like after breaking the input program down into tokens; `Parse` shows what the parse tree looks like after parsing the tokens; `Symbol-table` shows all of the defined identifiers in the input program and their properties; `Output-tree` shows the tree containing the elements of the input program, that is used in the implementation of the non-deterministic part of the compiler; and finally `All Models` shows all models generated from the compiler.

A.6 Grammar

Here follows the complete and exact grammar of the ESRA language used in the ESRA application. It is this grammar that one needs to follow when writing one's programs in the user interface. Before the grammar there is an explanation of the syntax used in the grammar.

```
Syntax:
<Type>     - entities in cursive inside < and >
             are non-terminals
int        - entities in plain text are
             terminals
{<Decl>}   - zero, one, or several times the
             entity inside { and }
<Constr>*  - one or several times the entity
             suffixed by the star, and where
             entities are separated by
             semi-colons (;).
-----------------------------------------------
Grammar:
<Model>      ->  {<Decl>}
                 <Instr>
<Decl>       ->  <DataDecl> ;
             ->  <VarDecl> ;
<DataDecl>   ->  <Type> <Id>
<Type>       ->  int
             ->  { int }
<VarDecl>    ->  var <TypeVar> <Id>
<TypeVar>    ->  <Id> -> <Id>
<Expr>       ->  <UnOp> <Expr>
             ->  <Expr> <BinOp> <Expr>
             ->  <AggrOp> ( <Param> ) <Expr>
             ->  <Argument>
```

```
                ->  ( <Expr> )
<UnOp>          ->  + | - | card
<BinOp>         ->  + | - | *
<AggrOp>        ->  sum | min | max
<Argument>      ->  <Object>
                ->  <Inverse>
<Object>        ->  <Id> | < <Id> , <Id> >
<Inverse>       ->  ~ <Id> [ <Id> ]
<Relation>      ->  <Expr> <ArithOp> <Expr>
                ->  <Expr> <SetOp> <Expr>
                ->  not <Relation>
                ->  <Relation> <LogicOp> <Relation>
<ArithOp>       ->  = | >= | <= | > | < | <>
<SetOp>         ->  in | not in
<LogicOp>       ->  & | \/ | => | <=>
<Constr>        ->  <Relation>
                ->  forall ( <Param> ) <Constr>
                ->  subset ( <Id> , <Id> )
                ->  injective ( <Id> )
                ->  { <Constr>* }
<Param>         ->  <Object> in <Bounds>
<Bounds>        ->  <Argument>
<Instr>         ->  solve <Constr> ;
                ->  minimize <Expr>
                    subject to <Constr> ;
                ->  maximize <Expr>
                    subject to <Constr> ;
```

The reader should note two agreed-upon restrictions that the grammar has. First, by the grammar rule of the *Object* non-terminal, n-tuples may contain at most *two* elements. Thus, the following ESRA statement is not valid:

```
for (<i,j,k> in F) i+j+k=3;
```

Second, by the grammar rule of the *Inverse* non-terminal, the inverse image of a mapping can be over only *one* element. Thus, one cannot write `~F[i,j]`.

A.7 Semantic Restrictions

The ESRA program not only needs to be grammatically correct, but also needs to follow certain semantic rules. There are two main rules:

1. Identifiers that are used in the ESRA program must be declared in the declaration part. The exception are the identifiers produced by the grammar rule

```
<Object>    ->  <Id> | < <Id> , <Id> >
```

when they are used as iterating identifiers inside a forall constraint or a sum expression. For identifiers that are used in the declaration part of the ESRA program to help declare other identifiers, their place in the declaration part where they are used is allowed to come before the place in the declaration part where they are declared.

2. Depending on where the identifiers are used, they must have the correct type. Listed below are the grammar rules that have such type requirements. The type requirements for each rule are specified after the vertical bar (|).

```
<Inverse>   -> ~ <Id> [ <Id> ]
             | the first identifier must be a
               mapping variable
<Constr>    -> injective ( <Id> )
             | the identifier must be a mapping
               variable
<TypeVar>   ->  <Id> -> <Id>
             | this rule is used for mapping
               variable declarations. Both
               identifiers must be declared as
               sets of integers
```

The compiler generates errors if any of these rules are broken: If rule number 1 is broken the semantic error *undefined identifier* is generated; if rule number 2 is broken the sematic error *invalid type* is generated.

There are other semantic issues worth noticing. Some ESRA programs, even though not generating any errors in the compiler, produce output programs that are conceptually incorrect.

First, there is the matter of ESRA programs using multiple mapping variable declarations. If more than one mapping variable is declared, the output programs produced by the compiler will not be the correct ones. The reason for this is described at the end of section 4.5 in the report.

Second, ESRA programs that contain two or more inverse image expressions, or one or more inverse image expressions and one or more sum expressions, do not work correctly with the compiler. The compiler will produce output programs that contain duplicates of one of the variable declarations. Section 4.5 in the report explains why.

Even though these issues can be viewed as errors, it has been agreed on that my targeted ESRA application is not required to resolve them.

Appendix B

ESRA Application Programmer's Manual

B.1 Introduction

This manual pertains to the second version of the ESRA application, i.e., that of phase 2, and is for anyone who wants to modify the code of the ESRA application. It gives the reader an overview and basic understanding of the implementation. Other documentation is also available: the reader can look in the files themselves for comments and also look at the javadoc generated documentation (`ESRA2/javadoc/index.html`). Note that the manual assumes that the reader is familiar with the contents of the report. A diskette containing the ESRA application in question is available together with the report.

B.2 Basics

The main and outermost directory of the ESRA application is the `ESRA2` directory. One get this directory when one decompresses the `esra2.tar.gz` file from the diskette. It contains, for example, the JLex application classes, the JavaCUP application classes, javadoc pages and other general files that are associated with the ESRA application.

The main code of the ESRA application, which is in Java, exists in the `ESRA2/esra` directory. This directory contains all the Java classes that I have implemented for the ESRA application. It represents a package structure with the following packages: `esra.userinterface`, `esra.utilities`, `esra.compiler`, `esra.compiler.tokenlist`, `esra.compiler.parsetree`, and `esra.compiler.symboltable`. The package `esra.userinterface` contains classes associated with the user interface, for example windows and dialog boxes; the package `esra.utilities` contains general classes with useful methods used by other classes, similar to `java.util`; `esra.compiler` contains files and classes

associated with the compiler core, for example the specification files for the tokenizer and the parser, and the rule file; and finally the package `esra.compiler.tokenlist`, the package `esra.compiler.parsetree`, and the package `esra.symboltable` are used for creating the text representations of the tokens, parse tree, and symbol table that one sees in the user interface.

Besides Java, Perl is also used in the ESRA application — the programs `tokenlist.pl`, `parsetree.pl`, and `ruleconv.pl` in the `ESRA2` directory are all in Perl. They represent the token list generator generator, the parse tree generator generator, and the rule converter program respectively.

To compile the whole ESRA application, type `esrac` at the command-line. This runs the `esra.compile` script in the `ESRA2` directory, which executes a series of commands — it uses the token list generator generator and the parse tree generator generator to convert specification files to new specifications files, JLex and JavaCUP to convert specification files to Java, the rule converter program to convert the rule file to Java, and javac to compile all the Java classes. Other useful aliases are `esrad` that updates the javadoc pages, and `esraz` that compresses the whole ESRA application so it fits on a diskette. All these aliases and the rest of the files are described more in the next section.

B.3 The ESRA2 Directory

The ESRA application is built up of a directory structure with several files and classes. In the following four sections I try to give a brief description of most of these files, classes and directories. To find the description for a file, class or directory, look for its name either in the title of a paragraph or inside a paragraph.

This section describes the main directory of the ESRA application, ESRA2. If one looks inside this directory one sees that it consists of several files and directories. Here is a description of this content.

Aliases I have written aliases for most of the common shell operations used with the ESRA application. For example, there is an alias for running JavaCUP called `jcup`, and there is an alias called `esraz` for compressing the whole ESRA application so it can be copied to a diskette. The file `aliases` contains all these aliases, as well as comments to what the different aliases do.

Javadoc As one can see by the javadoc comments in my Java source files, I am using javadoc in this project. The generated html files are put in the `javadoc` directory. The command file `esra.javadoc` contains the command for running javadoc on the `esra` packages. There is also an alias called `esrad` that one can use — it simply invokes the command file. Neither the command file nor the alias takes any arguments. The file *`packages`* is referenced from within the `esra.javadoc` file. Instead of writing all the esra packages in the javadoc command, they are put in this file.

The presed file The `presed` file defines a shell command that inserts a piece of text at the beginning of a file. It is called `presed` because it uses the command `sed` and puts the text before anything else in the file. I needed this command to insert a Java package declaration at the top of a Java source file. For example, the `lex.java` file, that is generated by the token list generator generator, needs to have the code string "`package esra.compiler.tokenlist;`" inserted at the top of it. All these invocations of `presed` occur within the `esra.compile` file.

ruleconv.pl, parsetree.pl, and tokenlist.pl These Perl files correspond in turn to the rule converter, the parse tree generator generator, and the token list generator generator, which are all described in my report. Like most other files in this project, they contain extensive documentation that one can also read.

JLex and JavaCUP The JLex application and the JavaCUP application are kept in the `JLex` directory and the `java_cup` directory respectively. To run JLex and JavaCUP there are aliases defined in the aliases file. If one needs to download these applications again for some reason, they can be found at their home pages: `http://www.cs.princeton.edu/~appel/modern/java/JLex/` and `http://www.cs.princeton.edu/~appel/modern/java/CUP/`.

esra.compile This is the shell script that I mentioned in the beginning of the manual, see section B.2, that compiles the whole ESRA application. It is this shell script that gets invoked when one executes the `esrac` alias. It not only uses `javac` to compile Java classes, but also invokes JLex and JavaCUP, as well as the token list generator generator, the parse tree generator generator, and the rule converter program. Whenever one makes a change in one of the specification files or update the rewrite rules, one should execute this shell script.

When this script runs, the following happens: `tokenlist.pl` converts `esra.lex` to `tlist.lex` in the `esra/compile/tokenlist` directory, `parsetree.pl` converts `esra_pass1.cup` to `ptree.cup` in the `esra/compile/parsetree` directory, and `ruleconv.pl` converts `EsraConverter.rul` into `EsraConverter.java` in the `esra/compiler` directory; then JLex and JavaCUP are used to convert the specification files, including the ones just generated, to Java classes; and finally all Java classes, including the ones just generated, are compiled with `javac`.

IMPORTANT NOTE: For this script to run properly, for example by typing `esrac`, the right file and directory permissions must be set. This is because the compile script uses the `cp` command to copy files within the `ESRA2` directory. If you get an error running this script, you most likely need to change the permissions of the directories and files in the `ESRA2` directory.

esra.run To start up the user interface (or the ESRA application, whichever way one views it) this shell script can be called. It basically calls the `Main` class, but also sets an option to the `java` command, which is needed to know which

directories contain the example files. There is also an alias called `esra` that is more commonly used.

esra.compress This shell script makes a tarred zip file of the `ESRA2` directory, which fits easily on a diskette. One can also call the `esraz` alias.

The readme file The `README` file contains simple instructions on how to install and run the ESRA application.

Main.java (and Main.class) This is just a very small Java class that starts the Java application by invoking the `MainWindow` class in the `esra.userinterface` package. The `Main` class itself is invoked by the `esra.run` script.

esra_files and opl_files These directories contain example programs, which one can access and use through the user interface.[1] The user interface is set to open these directories when the user chooses `open` from the file menu. The setting is done in the `esra.run` file.

The esra directory This is the directory that corresponds to and contains the Java package hierarchy of classes. I described this directory in the beginning of the manual, see section B.2.

B.4 The Compiler Directory

The Java classes that reside in this directory make up the `esra.compiler` package. Besides the Java classes, the lex and cup files are here, as well as the rule file.

Error handling I have made different exception classes for the different kinds of errors that can occur in the compiler. When there is an error in the tokenization, the exception `TokenizingError` **gets** thrown from the `Yylex` class, which is generated from the `esra.lex` file; when there is an error in the parsing, the exception `ParsingError` gets be thrown from the `esra_parser_pass1` and `esra_parser_pass2` classes, which are generated from the `esra_pass1.cup` and `esra_pass2.cup` files; when there is an error in the translation, a `SemanticError` gets thrown, also from the parser classes. All these three exception classes inherit from the `CompilerError` class. It contains a generic method for converting the exception to the formatted text representation of the error, that one sees in the user interface.

[1] The name, `opl_files`, is a little misleading as the directory does not contain OPL programs, but rather l-language programs. However, I use this name because it makes the application consistent with the application from phase 1.

The SemanticError class also has two child classes: InvalidTypeException and UndefinedIdentifierException. They are used mostly in the EsraConverter class that is generated from the rule file, EsraConverter.rul. The rule converter program generates Java code that throws those errors if needed. In the case of UndefinedIdentifierException, it is thrown if one of the parameters sent to the rule represents an identifier that has not been declared. In the case of InvalidTypeException, it is thrown if none of the translation alternatives in the rule matches.

The symbol table The result from the first parser pass gets stored in an object of class SymbolTable. The symbol table is implemented as a hash table in which identifiers, represented as strings, map to objects of class SymbolData. This class contains the data about the identifier, for example what type of identifier it is. These possible types are defined as static integer constants in the SymbolType class, and are for example INTEGER, TUPLE, MAPPING, etc.

SymbolInfo This class is described in the report, section 3.6. As it says there, this class is used to be able to more accurately report errors generated from the parser. Each generated token from the lexer gets associated with a SymbolInfo objects, which holds information on which line, which column, etc., the token occurs in the input text.

The tokenizer classes esra.lex is the JLex specification for the ESRA language. In the compile script it gets converted to esra.lex.java, which in turn gets compiled to the Java class Yylex. Read more about how this works in detail in the JLex manual, see [1].

The parser classes esra_pass1.cup and esra_pass2.cup are the first and second pass parser JavaCUP specification files. From these two files, the compile script uses JavaCUP to generate esra_parser_ pass1.java and esra_parser_ pass2.java; and two copies of sym.java. The two parser Java files are then compiled into an action class each: CUP$esra_parser_ pass1$actions.java and CUP$esra_parser_ pass2$actions.java, as well a parser class each: esra_ parser_pass1.class and esra_parser_pass2.class. The sym.java file is compiled into sym.class. The reader can read more about how this works in the JavaCUP manual, see [2].

Grammar helper classes Some of the non-terminals in the ESRA grammar that occur in the cup specification files are also represented as Java classes with the same name. These classes are Type, TypeVar, Object_, and Parameter (note that for the non-terminal Object, the class name Object_ had to be used, since Object already exists in Java). Normally in the JavaCUP specification file, the result from parsing a non-terminal is of type String, representing the translation of that non-terminal. However, for some productions one needs to have information about some non-terminal in order to make the translation.

Then it is important that we do not make the translation right away, but wait a couple of steps up the parse tree. In this case, a Java object and not a string, is passed through the label of the non-terminal. The Java object (`Type`, `TypeVar`, `Object_` or `Parameter`) contains the necessary information about this non-terminal, so that the translation can be made later on.

Tree classes The class `OutputTree` represents the element tree, which makes the non-deterministic translation possible by generating a set of different output models. It is built up recursively of `TreeNode` objects, and these contain objects of class `Element`, class `ElementSet`, or class `ElementSetChoice`. An `Element` object also contains an `Explanation` object. For a more thorough description of the classes, read sections 4.2 and 4.3 of the report.

YylexWrapper In the `parsetree` directory there is a class called `ParseTreeGenerator` that handles the process of calling and combining the classes that are generated from the parse tree generator generator. It needs access to the `Yylex` class in this directory. Since `Yylex` is not public (I tried and tried to make it public using different options to JLex, but couldn't), and the `ParseTreeGenerator` class is in another package, it doesn't have the right to access it. However, by wrapping the `Yylex` class inside a public class, `YylexWrapper`, it is now accessible. The wrapping is done by simply letting `YylexWrapper` inherit from `Yylex` and recreating its two constructors. The two new constructors simply invoke the two old constructors.

Rewriter This is a small class that connects the lexer and the two parsers together by handling all the calls between the classes. It has the same function as the `SymbolTableGenerator` class in the `symboltable` directory, the `TokenListGenerator` class in the `tokenlist` directory, and the `ParseTreeGenerator` class in the `parsetree` directory. The `Rewriter` class is also described in the report, see section 3.5.

Compiler This class serves as an API to the user interface. All communication from the user interface to the compiler go through this class.

Model and ModelDatabase Every model extracted from the tree gets represented as a `Model` object. It basically contains lists of the different kinds of elements that make up its program. Its most important method is the `toString` method that converts the `Model` object into a natural looking program. `ModelDatabase` is simply a list, which holds all the models generated from the tree. Its `toString` method prints out all models in a nicely formatted way — it is used by the user interface when the user chooses `all models` from the menu.

LowerCaseModule This is a very small class that can convert all big characters in a text ('A', 'B', 'C') to small characters ('a', 'b', 'c'). It was originally used to convert all ESRA programs to lower case, but now it is no longer used. The user, thus, is required to write all key words with small letters, which anyway is the case with most other programming languages. Note that I've decided not to delete this class, because it can be useful in the future.

The rule file `EsraConverter.rul` holds all the rewrite rules. In the compile script the `ruleconv.pl` program is used (with the `rulcon` alias) to convert it to `EsraConverter.java`.

The tokenlist directory The classes in this directory make up the `esra.compiler.tokenlist` package. I needed to make a separate directory for the token list generator functionality — otherwise the `Yylex` class of the the normal lex file would clash with the `Yylex` class of token list generator lex file. In the compile script the `esra.lex` file is copied from the compiler directory into this directory. By using the `tokenlist.pl` program in the `ESRA2` directory, `esra.lex` is converted into `tlist.lex`. Using JLex, `tlist.lex` is converted into `tlist.lex.java`, and in turn compiling this Java file gives two classes: `Yylex` and `Yytoken`. Instead of, as normal, using the `Symbol` class for representing tokens, we now use the `Yytoken` class for representing tokens. This is because we now also want to store the name of the integer constant as a string: for example, for `sym.INT` the string "`INT`" is stored in the token. With this information the generated token list generator can properly display the list of tokens. Finally, the class `TokenListGenerator` ties everything together and handles the calls between the different classes.

The parsetree directory This works similarly to the `tokenlist` directory. `esra.cup` is copied from the compiler directory, then converted to `ptree.cup` using `parsetree.pl` in the `ESRA2` directory, which in turn is converted to `parser.java` and `sym.java` using JavaCUP. Compiling these Java files gives `parser.class`, `sym.class`, and `CUP$parser$ actions.class`. Finally, `ParseTreeGenerator` connects everything together.

The symboltable directory For printing out the symbol table a separate directory was not really needed. However, it looked more symmetrical if this functionality, like token list generator and parse tree generator, also had its own directory. In this directory there is only one class, `SymbolTableGenerator`, which calls the `toString` method in the `SymbolTable` class to print out the symbol table in the user interface.

B.5 The User Interface Directory

This directory corresponds to the `esra.userinterface` package.

MainWindow The main class in this directory is the `MainWindow` class. It represents the window object that gets opened when the user starts the application. It contains code for handling of menu events, handling of the second window that shows the compilation result, handling of the different dialog boxes that sometimes pop up, and handling of the file dialog box.

Dialog Windows There are three different dialog messages that can occur in the application: "`This file does not exist, do you want to create it?`", "`The file is not saved, do you want to save it?`", and "`This file already exists, do you want to overwrite it?`". They all are instances of the `YesNoDialog` class, which has one yes button, one no button and one cancel button. There is also another similar class called `OkDialog` that has one ok button and one cancel button. It is used when an error in the application occurs, for example when creating a file in the file system fails. I have never seen it happen, but it could, for example, if the file permissions are set improperly. Both `YesNoDialog` and `OkDialog` inherit from `MyDialog` that holds some shared code, for example a method for centering the dialog box — both the `YesNoDialog` box and the `OkDialog` box needs to do this.

MyFileDialog One might ask why I have made this class when the class `FileDialog` already exists in Java. The reason is that the `FileDialog` class is a little clumsy to use. With `MyFileDialog`, which inherits from `FileDialog`, I've added the method `userChooseFile` which handles, for example, the user pressing cancel, the user leaving the text field empty, and the storing of a *most recent visited directory* entity.

CompileWindow Finally, there is the `CompileWindow` class, which represents the second window that gets opened whenever the user chooses an action from the compile menu. It contains code for handling its only two menu choices: `save as` and `close`, for expanding the window horizontally when the user chooses 'output tree' from the menu, and for positioning the window in the right place next to the main window.

B.6 The Utilities Directory

This directory represents the `esra.utilities` directory.

Formatter This is probably the most important class in this package. It contains a set of static methods that are used throughout the application. The two main methods are `tab` and `blockify`. `Tab` can indent every line in a string by a specified number of characters and is used in the code that generates the OPL program, so that the OPL code gets formatted nicely. `Blockify` can take a long string and break it into several lines of approximately equal length, though not breaking any words in two, and is used to present the error message in the compile window, so that it doesn't mess up the nice looking tabular format.

TextSubstituter Like `Formatter` this class contains only static methods. By far the most important method in this class is `substituteId`, which is capable of distinguishing identifiers in a long string of text and replacing a specific one with a new string entity. For example, it replaces `foo` in `foo+3`, but not `foo` in `foot+3`. The method is used with the rule file converter for handling the substitution expressions, `[: / :]`, that can occur in the rule file, see the report, section 4.4.

MyList This is also a class not meant to be instantiated. It has only one method, `listToString`, which can convert a list into a string, but this method was mainly used in the first phase of the ESRA application, where `forall` and `sum` expressions could take multiple parameters.

MyFile `MyFile` is similar to the `MyFileDialog` case. It is an extension of Java's `File` class and adds the two useful methods `readToString` and `writeFromString`.

B.7 Flow of Execution

I now describe what happens when running the ESRA application, i.e., what the basic flow of the program looks like.

From the Main class to the Rewriter class Firstly, when the user types `esra`, the `Main` class in the `ESRA2` directory gets called. It in turn calls the `MainWindow` class in the `esra.userinterface` package that displays the actual window. When the user has written a program in the window and presses one of the commands in the compile menu, methods with similar names get called in the `Compiler` class in the `esra.compiler` package. Then, for each of the commands, the `Compiler` class calls the `generate` method of the classes with corresponding names: `TokenListGenerator` in the `esra.compiler.tokenlist` package, `ParseTreeGenerator` in the `esra.compiler.parsetree` package, `SymbolTableGenerator` in the `esra.compiler.symboltable` package, and `Rewriter` (used for both the commands `Output-tree` and `All Models`) in the `esra.compiler` package.

Inside the Rewriter class The `generate` methods of the four classes mentioned just above do approximately the same thing. I will focus on the `Rewriter` class and the case when the user chooses the command `All Models` and describe the flow from there. The `Rewriter` class, or more correctly its `generate` method, starts by creating a `Yylex` object that takes a stream of the input program as argument. It then creates an `esra_parser_pass1` object and feeds the `Yylex` object to it. The `parse` method of the `esra_parser_pass1` object is called and the semantic code in the object executes over the input program and returns a `SymbolTable` object.

Next, a esra_parser_pass2 object is created with the SymbolTable object as argument to its constructor. Its parse method is called and the semantic code executes over the input program and returns an OutputTree object. It then calls the generateAllModels method of the OutputTree object which converts the tree to a ModelDatabase object. The generateAllModels method uses a recursive tree algorithm to insert Element objects into Model objects that are inserted into the created ModelDatabase object. The ModelDatabase object is then converted into a displayable format — a String object — by calling its toString method. This toString method in turn calls the toString methods of the Model objects that convert the Model objects to displayable programs.

The generate method has hereby finished its execution and returns the String object to the call made from the Compiler object, which in turn returns the String object to the call made from the MainWindow object. The MainWindow calls setBuffer in the CompileWindow so that the string — the translated result, the displayable programs — get shown in the second window.

Inside the esra_parser_pass2 class Above I mentioned just briefly that the esra_parser_pass2 class translates the input program to the output program. Here next, I describe what actually happens inside this class in more detail.

To begin with, the semantic code of the production rules gets executed over the input program. Some of the productions translate to themselves so a string result is returned with the same string that got parsed. For other productions, like forall, sum, variable declarations, etc., the translation is more complex. For these productions, a call is made to a method with a similar name in the Java class EsraConverter.java that corresponds to the rule file EsraConverter.rul. The arguments to this method are objects of type Object that can be either SymbolInfo or ElementSetChoice. ElementSetChoice means that the parameter represented by this object already has several representations — this is the case for, for example, expressions.

The translation is then made inside the rule file method and the result is returned in form of a new ElementSetChoice object. The ElementSetChoice object is then either returned by the semantic code to its parent production, or inserted into the outputTree object using its addElementSetChoice method, which uses a complex algorithm defined in the method addToAllLeaves to insert the choice properly into the tree.

For the production rules there is also the case when one or more of the sub results from the labels are ElementSetChoice objects. The production rule then needs to combine these choices in some way. It does this by calling the concat methods that are defined in the ElementSetChoice class. The concat methods use combinatory algorithms to combine the choices.

B.8 Example

Here, I show an example of updating the programming code of the ESRA application. This example is about adding a new statement to the ESRA language.

The problem The statement that we want to add is a constraint that looks like `foo(F)`. It is a function with the name `foo` that takes one argument, `F`, which must be a mapping variable. The following is the translation rule for the `foo` constraint — it uses one of three representations depending on which representation `F` is using.

```
foo(F), where F is a mapping var. from V to W
rep 1) comment: check that all are 1's.
       forall(i in V) F[i] = 1;
rep 2) comment: check that the sum is zero
       sum(j in W) F[1,j] = 0;
rep 3) comment: W_range is a subset of V
       var {V} W_range;
       comment: all elements make up the range
       union all(i in F[i]) = W_range;
```

Changes to the tokenizer First we need to affect the tokenizer. We need to make `foo` a new keyword of the language. To do this we open up the `esra.lex` file in the `compiler` directory, and insert the following code with the other token definitions:

```
<YYINITIAL> "foo" { return new
  Symbol(sym.FOO,new SymbolInfo(
    yytext(),yyline,linechar(),yychar));
}
```

Changes to the parser Next we need to affect the grammar of the language. The grammar is defined in the cup specification. As our compiler is a two pass compiler, it has *two* specification files. Both contain the same grammar but have different semantic actions. We therefore need to modify both specification files, and these are `esra_pass1.cup` and `esra_pass2.cup` in the compiler directory.

We start with the `esra_pass1.cup file`. In it we need to declare `FOO` to be a non-terminal of the language. We do this near the top of the file. There one can see a chunk of terminals declared there that look like this:

```
terminal SymbolInfo CARD,X,VAR;
terminal SymbolInfo FORALL,IN,INT,MAX,MAXIMIZE;
terminal SymbolInfo MIN,MINIMIZE,NOT;
...
```

We add `FOO` to one of the lines, for example like this:

```
terminal SymbolInfo MIN,MINIMIZE,NOT,FOO;
```

The reader might wonder why we need both a definition of the key word `foo` here in the cup file and in the lex file. The answer is that what we declared in the lex file was that the grouping of characters, 'f', 'o', 'o', should be a token and correspond to the `FOO` constant defined in the sym class. For this `FOO` constant to at all exist in the sym class, we need to define it somewhere, and that is what we just did, here in the cup file.

Then we scroll down to the production rule for the `Constraint` non-terminal, the one that looks like this:

```
Constraint ::= Relation
  | FORALL LPAREN Parameter RPAREN Constraint
  | SUBSET LPAREN ID COMMA ID RPAREN
  | INJECTIVE LPAREN ID RPAREN
  | LCURLY Constraint RCURLY
  | LCURLY Constraint Constraints RCURLY
  ;
```

`Relation`, `FORALL..`, etc., are different production alternatives. We add the new production alternative:

```
  | FOO LPAREN ID RPAREN
```

Like the other production alternatives here, it does not need any semantic code in this first cup file.

Next we open up the second cup file, `esra_pass2.cup`. Like the first cup file, it has a section of terminal declarations at the top of the file, so we insert `FOO` into it.

Then we are going to affect the production rules. Even though the grammars in the two cup files produce the same language, the arrangement and naming of the rules are a little different. As the first cup file has one rule for constraints named `Constraint`, the second cup file has two rules for constraints named `Constraint` and `BaseLevelConstraint`. `Constraint` refers to the constraints that are subparts of other constraints, like a `forall` constraint nested inside another constraint, while `BaseLevelConstraint` refers to the constraints at the outermost level in the solve (or minimize of maximize) clause. As `foo` cannot be nested inside another constraint, we need to affect the `BaseLevelConstraint` rule and not the `Constraint` rule. (If the reader wonders why we at all need to distinguish between these two types of constraints, the answer is that they have different semantic actions, for example only constraints of type `BaseLevelConstraint` should be inserted into the tree, not constraints of type `Constraint`).

So now that we know in which rule to insert the `FOO LPAREN...` production alternative, we need to determine what the semantic action should be. It is the semantic action that should handle the translation. Since this is a kind of complex translation (it has several representations), we should put the definition

of the translation rule in the rule file. In the semantic code we will only refer to the definition in the rule file. This is what it all will look like in the cup file:

```
| FOO LPAREN ID:id RPAREN
{: ElementSetChoice c =
     esraConverter.fooConstraint(id);
   outputTree.addElementSetChoice(c);
:}
```

The effect of the first line is that the rule in the rule file needs to be called `fooConstraint` and take one argument. As it returns a set of different representations, the return type is `ElementSetChoice` (it is always `ElementSetChoice`, even for those translation rules that only have one representation). The second line inserts the set of representations, `c`, into the tree.

Changes to the rule file The only thing that remains now is to create the rule in the rule file. One does this by opening up the file, `esraConverter.rul`, in the compiler directory, and putting the following code somewhere in the file:

```
cons fooConstraint(F) {:
    =>   % check that all are 1's.
         forall(i in V) F[i] = 1;
     |   child:1;F:varmap(V->W)
    =>   % check that the sum is zero
         sum(j in W) F[1,j] = 0;
     |   child:2;F:varmap(V->W)
    =>   % W_range is a subset of V
         var {V} W_range;
         % all elements make up the range
         union all(i in F[i]) = W_range;
     |   child:3;F:varmap(V->W)
```

The first word `cons` has actually no purpose. It did in the first phase of the ESRA application, but not any longer. However, one still needs to have it there (or optionally `decl` or `expr`), because otherwise the rule file does not parse correctly.

Then follow the different translation alternatives, each starting with an arrow and consisting of one or several statements and an ending condition. A single statement can span several lines, so a semi-colon is needed to indicate the end of a statement. Each statement, as seen above, can also have a comment associated with it. The comment should be placed directly above the statement, and the line or lines that make up the comment should start with a percent (`%`) sign. The condition at the end is distinguished by it being on its own line that starts with a vertical bar (`|`). In our example the conditions generally say that `F` must be a mapping variable. The `(V->W)` part lets us extract the domain and the codomain of the mapping and refer to these entities in the statements. Each translation alternative also has a `child` declaration in the condition. It means

that the translation alternative follows, not sets, a representation. The number after the colon denotes the representation number, the unique key that identifies the representation.

For more details about the rule file, read the extensive documentation at the top of the rule converter program, `ruleconv.pl`, in the `ESRA2` directory. The reader might also find section 4.4 of the report useful.

Conclusion That completes the example. Type `esrac` to compile the new changes and then `esra` to launch the user interface. To test that our changes work, type in the following ESRA program and try compiling it:

```
{int} V;
{int} W;
var V->W F;
solve {
  foo(F)
};
```

Bibliography

[1] Elliot Berk. JLex user manual. *JLex: A lexical analyzer generator for Java.* 1997. A link to the manual exists on the JLex homepage at URL: http://www.cs.princeton.edu/~appel/modern/java/JLex/

[2] Scott E. Hudson. *CUP User's Manual.* 1999. The user manual can be found at the homepage of JavaCUP at URL: http://www.cs.princeton.edu/~appel/modern/java/CUP/

[3] Kim Marriott and Peter J. Stuckey. *Programming with Constraints, An Introduction.* The MIT Press, 1998.

[4] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. *Compilers; Principles, Techniques and Tools.* Addison-Wesley Publishing Company, 1986.

[5] Pascal Van Henteryck. *OPL; Optimization Programming Language.* The MIT Press, 1999.

[6] Pierre Flener and Brahim Hnich. *The Syntax and Semantics of ESRA.* ASTRA report, March 2001. Available via http://www.csd.uu.se/~pierref/astra

[7] Brahim Hnich. *Function Variables for Constraint Programming.* PhD Thesis. In preparation.

[8] Pierre Flener, Brahim Hnich and Zeynep Kiziltan. *Compiling High-level Type Constructors in Constraint Programming.* Available via http://www.csd.uu.se/~pierref/astra