UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft
Fakulät für Elektrotechnik, Informatik und Mathematik

Bachelor Thesis

Integrating a GLR Parser Generator in Eli

Ulf Schwekendiek
Matrikelnummer: 6300228
E-Mail: sulf@uni-paderborn.de

Paderborn, den 7. August 2007

vorgelegt bei
Dr. Peter Pfahler

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Paderborn, den 7. August 2007

Ulf Schwekendiek

Danksagung

Ich bedanke mich an dieser Stelle von ganzem Herzen bei Christina. Du gabst mir Liebe, Unterstützung und immer neuen Mut. Genauso bedanke ich mich bei meinen Eltern, die immer an mich geglaubt haben und hinter mir standen.

Contents

1 Introduction

Eli is a toolbox to design and implement programming languages and application specific languages. It contains tools and libraries which help to automatize the development process of a compiler.

Eli includes a scanner generator, called Gla, two different LALR(1) parser generators which are called Pgs and Cola and an attribute analyzer generator called Liga. Eli is based on an expert system, called Odin, which helps the user to handle all the included tools in a more efficient way.

The goal of my thesis is to add a third parser generator, called Bison to Eli. Bison lies under the GNU General Public License and is a general purpose parser generator. It converts annotated context free grammar into a LALR(1) or GLR parser. It is complete upward compatible with AT&T's YACC. Bison was written by Robert Corbett and it was Richard Stallmann, who made it Yacc compatible. Because Bison generates parsers that can handle a larger set of grammars, it would be a big advantage to add Bison as an additional parser generator to Eli.

1.1 Goals

Parsers that accept strings defined by LALR(1) grammars like Pgs and Cola are deterministic. Figure 1.1 shows the hierarchy of grammars. As you can see from the picture Pgs and Cola only generate parsers for up to LALR(1) grammars. On the other hand, Bison's GLR parsing technique generates parsers capable of handling grammars ranging in complexity from LR(0) to context-free. For Eli the grammar designer has to make sure that he meets the properties for a LALR(1) grammar. With Bison he would gain a lot more freedom in writing grammars. Therefore it would be very useful to add Bison as a package to the Eli system.

In Bison it is possible to toggle from LALR(1) to Generalized LR (GLR) parsing. With this technique the parser is able to handle any context free grammar that has finite strings.

It is also important that Bison should not have any significant complexity increase. [10] showed, that the time required to run the algorithm is proportional

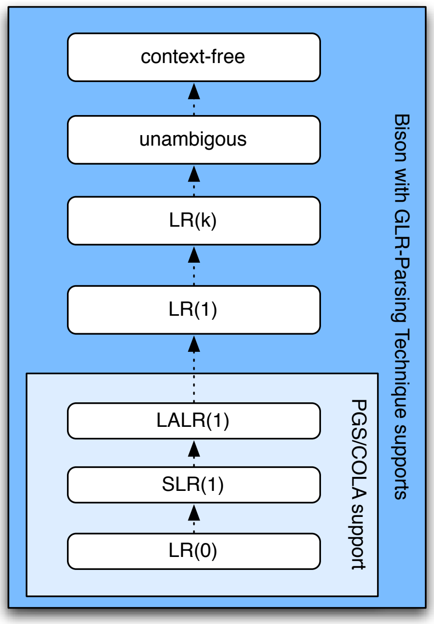


Figure 1.1: Parser Hierarchy

to the degree of nondeterminism in the grammar.

This thesis contains five main parts. In order to describe the integration of a new GLR parser into the Eli system, it is necessary to give an introduction to the basics of each element, that occurs in this thesis.

Hence the second chapter “Basics” covers an introduction to Bison, the GNU parser generator, Eli, the expert system in which Bison should be integrated and to the GLR parsing technique, which is used by Bison.

Because the Eli system consists of tool packages, it is important to define a new package for Bison and to adjust already existing packages to integrate Bison as a third parser generator, which is described in chapter 3.

In order to design this new package a fundamental part is the definition of a translator (chapter 4), which has to convert a grammar file into a Bison specification, which makes the use of Bison possible. This generated file contains both the interface for the attribute analyzer generator and the complete structure of the grammar.

The evaluation (chapter 5) shows several tests in order to verify the correct implementation of an GLR-capable parser and the successful integration into the

Eli system.

The last chapter resumes this thesis and discusses whether the use of Bison is an advancement or in which cases it is better to stick to Pgs or Cola.

2 Basics

2.1 Introduction to Bison

Bison is a general purpose parser generator. It converts annotated context-free grammars into LALR(1) or GLR parsers in C or C++. Bison uses a machine like version of the Backus-Naur form (BNF), which was developed in order to specify Algol 60.

At this point I will explain the basic concepts of using Bison in order to generate a parser. Here I will constrain my documentation on how to generate C parsers. However more general information on compiler construction can be found in [8] or [3]. For more information about how to generate C++ parsers see [6, pp. 101-110].

2.1.1 Layout of a Bison Grammar File

The input for the Bison parser generator is a Bison grammar file, which conventionally has the ending "`.y`". The general form of it is shown in example 2.1. In any parts of the file it is allowed to use C-like comments. These are comments enclosed in "`/* ... */`" or a one line comment started with "`// ...`".

Example 2.1 Example of a Bison semantic action

```
%{
Prologue
}%

Bison declarations

%%
Grammar rules
%%

Epilogue
```

Prologue

In the prologue the Bison user can define types, variables and actions in C or C++ notation depending on what the parser generator should produce. The prologue is surrounded by "`%{`" and "`}%`". At this part macros and `#include` directives, to include header files, can be used. The scanner interface function `yylex` and the error printer function `yyerror` have to be defined here. If there are no definitions necessary the Bison user can omit the prologue. The complete block is copied to the beginning of the generated parser source file, so that they precede the definition of `yyparse`, the parsing function. It is also allowed to have more than one prologue block. It allows the Bison user to have C/C++ and Bison declarations, that refer to each other (see [6, p. 41]).

Bison Declarations

In the Bison declarations part, an important thing is to define and declare the terminals (also called tokens), nonterminal symbols, and specify production precedence for resolving parse conflicts.

The declaration part is divided into:

- **Token declarations:**
  A token can be declared in different ways. After Bison is executed it can generate a header file, which can be used in the scanner. In this header file the "`%token`" declarations are converted to `#define` directives. Example 2.2 shows all different methods to declare one token.

Example 2.2 Example of a Bison token definition

```
1  %token INTEGER
2  %token IF, THEN, ELSE
3  %token FLOAT 5
4  %token GT 6 ">="
```

  The basic way to declare a token is "`%token` *name(s)*" (as shown in example 2.2 line 1). Bison automatically chooses an unique integer code for each token. Token codes can also be provided in the specification (see example 2.2, line 3). It is also allowed to list more than one token (see line two of example 2.2) and it is possible to use decimal or hexadecimal integer value for specifying the token. Bison does not allow to set the token code for a literal terminal. However it is possible to specify a character string after the token definition in double quotation marks. Important here is that this character sequence definition must be in the grammar in double quotation marks and not in single ones (see 2.2 line 4).

- **Types:**
  It is also allowed to define multiple types for terminals. That is possible with the "`%union { ... }`" declaration, which is also called "stack type". Example

2.3 shows such a multiple type definition with two different stack types, `intVal`

Example 2.3 Example of a stack type definition

```
1 %union { /* define stack type */
2    int intVal;
3    char   *stringVal;
4 }
5 %token <intVal> INT
6 %token <stringVal> STRING
```

and `stringVal`. In line 5 and 6 each token is assigned to one of them. The token type has to be written in angle brackets in front of the token name. If multiple types are used, they can also be assigned to nonterminal symbols. The declaration name for that is `%type` and used exactly like the `%token` declaration (see example 2.5).

Example 2.4 Notation of a type declaration for nonterminal symbols

```
%type <type> nonterminal ...
```

- **Operator Precedence:**
  There are three declarations that can be used: `%left`, `%right` and `%nonassoc`. The notation for that is shown in Example 2.4. These declarations are used for resolving parsing conflicts.

Example 2.5 Notation of a type declaration for nonterminal symbols

```
%left symbols ...          /* simple    */
%left <type> symbols ...   /* with type */
```

  For more information about operator precedence see [6, p. 73].

- **Start Symbol:**
  Bison assumes by default that the start-symbol is the left hand side nonterminal from the first production in the grammar. The Bison user is allowed to overwrite this restriction with the `%start <symbol>` declaration.

- **GLR:**
  The `%glr-parser` declaration changes the Bison parsing algorithm from LALR(1) to GLR and Bison generates now a GLR parser. I will explain the GLR technique in the next section of this thesis in detail.

A complete list of Bison definitions can be found in [6, pp. 59-61].

Grammar Rules

Nonterminal symbols are represented like C identifiers. By convention it should be in lower case such as “`start`” or “`expr`”. Terminal symbols are also represented as C-like identifiers. By convention, these identifiers should be in upper case like “`NUM`” or “`FLOAT`”. A literal terminal symbol like `’=’` or `’+’` can be represented as a character literal, just like a C character. A sequence terminal holds a character sequence and is surrounded by double quotes like `"print"`. Example 2.6 shows a Bison grammar, which includes all basic grammar rules. A production is terminated by a semicolon and with the dash another production with the same left hand side is defined. After a production a semantic action can be defined. There is one special production which has the “`error`” token included. For instance, line 9 in example 2.6 shows such a production. If in one of the productions with “`statement`” on the left hand side an error occurs all tokens until the semicolon are skipped and after that the parsing resumes.

Example 2.6 Bison grammar

```
1   /* GRAMMAR */
2   start        : stmtlist
3                ;
4
5   stmtlist     : stmtlist statement
6                | statement
7                ;
8
9   expr         : expr ’+’ expr  {$$ = $1 + $3;}
10               | expr ’-’ expr  {$$ = $1 - $3;}
11               | expr ’*’ expr  {$$ = $1 * $3;}
12               | expr ’/’ expr  {$$ = $1 / $3;}
13               | ’(’ expr ’)’   {$$ = $2;}
14               | NUM            {$$ = $1;}
15               | ’[’ NUM ’]’    {$$ = memory[$2];}
16               ;
17
18  set          : ’=’ expr       { $$ = $2; }
19               | /* empty */    { $$ = 0;  }
20               ;
21
22  statement    : "print" expr ’;’  { printf(">Result: %d\n", $2); }
23               | "set" NUM set ’;’ { if ($2 < MAX_MEM) memory[$2] = $3;}
24               | "end"
25               | error ’;’ { printf(">Unknown statement.\n"); yyerrok; }
26               ;
```

Semantic Actions

In Bison, actions can be associated with productions. Each production can have one action, which is executed after that production is parsed and can consist of multiple C statements. Most of the time, the purpose of such action is to construct an annotated parse tree or compute the semantic value of the production. An action can be defined in curly brackets everywhere on the right hand side of a production and can last over more lines.

Example 2.7 Bison semantic actions

```
expr :  expr '-' expr{ $$ = $1 - $3; }
     |  '(' expr ')' {
                          printf("parenthesized expr parsed");
                          $$ = $2;
                     }
     ;
```

Example 2.7 shows such a semantic action. `$$` represents the semantic value from the left hand side of the production and `$1`, `$2`, ..., `$`n represents the semantic value of the n-th symbol of the right hand side from a production. The "semantic value" is a single unnamed attribute of each nonterminal symbol.

Epilogue

Just like the prologue the epilogue is copied verbatim to the end of the generated parser source file. At this place it is convenient to place anything that the programmer wants in the parser source file, which does not need to come before the definition of `yyparse`, the parsing method. Because C requires functions to be declared before being used, it is important, that they are declared in the prologue before they are used in the epilogue.

2.1.2 Scanner Interface

Bison expects that a function `yylex` is defined and implemented. This function recognizes tokens from the input stream and returns them to the parser and should be defined in the prologue of the Bison grammar file. To let Bison generate a header file which contains the macros (for example: `#define <tokenname> NUMBER`) of the defined tokens, it has to be called with the "`-d`" option. The convention for `yylex` is, that it must return a positive numeric code for the type of token it has found. A zero or negative value shows the end-of-input. The scanner has also to make sure that the semantic value of a token is put into the variable `yylval`. If the textual location of tokens is important then the scanner has also set the global variable `yyloc`. By default, the datastructure of `yyloc`, `YYLTYPE`, is a structure explained in example 2.8.

Example 2.8 Default structure of YYLTYPE

```
typedef struct YYLTYPE
{
   int first_line;
   int first_column;
   int last_line;
   int last_column;
} YYLTYPE

YYLTYPE yyloc;
```

2.1.3 Debugging and Tracing the Parser

[6, p. 71] describe the parsing algorithm of the generated parser and document, that the generated parser is a shift/reduce automaton. There are two ways to inspect this automaton. The programmer can use the "`-report`" or "`-verbose`" flag to generated a `.output` file which contains this automaton and more debugging information such as resolved conflict states and useless nonterminal symbols.

The other way to show the automaton is the "`-g`" flag. With this flag Bi-

Example 2.9 Example grammar for generating the shift/reduce automaton

```
%{
  /* ... */
%}
%token NUM
%left '+' '-'
%%
exp:  exp '+' exp
   |  exp '-' exp
   |  NUM
   ;
%%
/* ... */
```

son generates a graph of that automata in form of a `.vcg` file, which can be used with a VCG-Viewer (see [1]). Figure 2.1 shows such a generated automaton for the grammar in example 2.9.

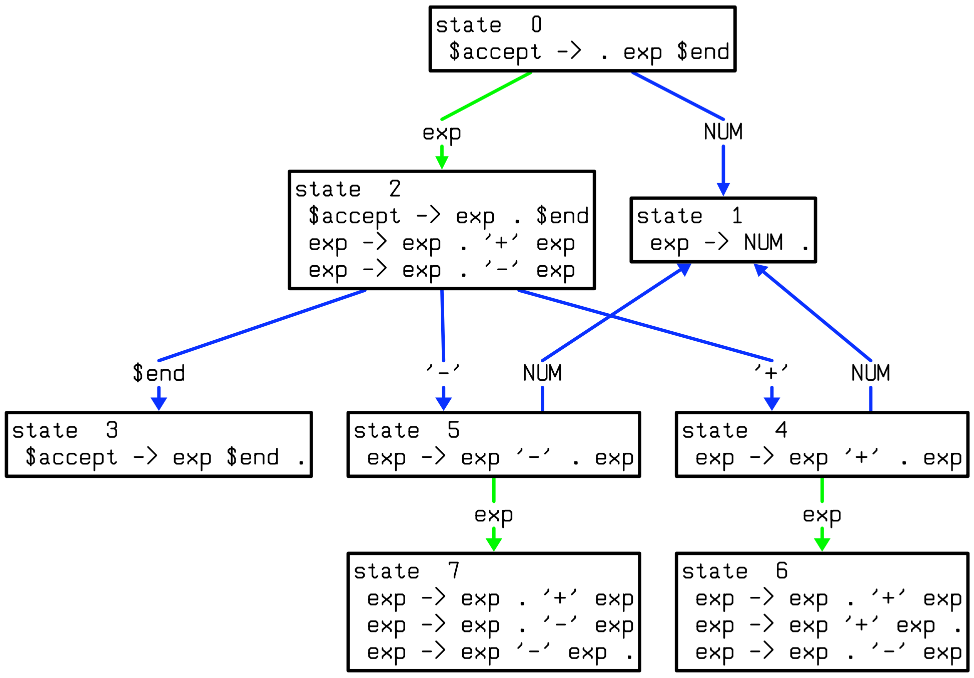


Figure 2.1: Generated Shift/Reduce automaton for example grammar 2.9

In order to generate the parser with trace and debug information the programmer has the choice of three ways to activate them. He can use the macro `YYDEBUG` and set it to 1 in order to activate tracing. He can also use the "`-t`" or "`--debug`" flag when Bison is invoked or he uses in the Bison definition part the definition "`%debug`" to activate tracing and debugging. For more information see [6, pp. 95-96].

2.1.4 Invoking Bison

The usual way to invoke Bison is "`bison` *`bison-grammar-file`*". The grammar file has by convention the ending "`.y`". Bison will generate the parser and name it ".tab.c". A very important flag is the "`-d`" flag. It generates in addition to the parser a header file "`.tab.h`" which should be used as an interface for the scanner. All possible flags are in [6, pp. 97-100].

2.1.5 Example Program

Here is an example for a Bison program. It generates a simple integer calculator and can load and store results to memory. I used Flex for generating the scanner and Bison for the parser:

Listing 2.1: calc.l - Scanner definitions

```
%{
#include <stdlib.h>
#include <string.h>
#include "calc.tab.h"
%}



%%

[;]|[+]|[-]|[*]|[/]|[(]|[)]|[\]]|[\[]|[=]   { return yytext[0]; }

[0-9]+        { yylval = atoi(yytext); return NUM; }
"print"       { return PRINT; }
"end"       { return EOF; }
"set"       { return SET; }
.         { /* skip unknown token */ }
[ \t\n]+        ;

%%
```

Listing 2.2: calc.y - Bison grammar file

```
%{
/* PROLOGUE */
#include <stdio.h>
#include <stdlib.h>

#define MAX_MEM 10

int yyerror(char *s);
int yylex(void);
extern FILE *yyin;

int memory[MAX_MEM];

%}

/* DEFINITIONS */
%start start

%token NUM
%token PRINT     "print"
%token END   666 "end"
%token SET     "set"

%left '+' '-'
%right '*' '/'

%%
/* GRAMMAR */
start       : stmtlist
            ;
```

```

stmtlist     : stmtlist statement
             | statement
             ;

expr         : expr '+' expr  {$$ = $1 + $3;}
             | expr '-' expr  {$$ = $1 - $3;}
             | expr '*' expr  {$$ = $1 * $3;}
             | expr '/' expr  {$$ = $1 / $3;}
             | '(' expr ')'   {$$ = $2;}
             | NUM            {$$ = $1;}
             | '[' NUM ']'    {$$ = memory[$2];}
             ;

set          : '=' expr       { $$ = $2; }
             | /* empty */    { $$ = 0;  }
             ;

statement    : "print" expr ';'  {
                 printf(">Result: %d\n", $2); }
             | "set" NUM set ';' {
                 if ($2 < MAX_MEM) memory[$2] = $3;}
             | "end"
             | error ';' {
                 printf(">Unknown statement.\n"); yyerrok; }
             ;
%%

/* EPILOGUE */
int yyerror(char *s)
{
  fprintf(stderr,"%s\n",s);
}

int main(int argc, char *argv[])
{
  if (argc > 1)
  {
    yyin = fopen(argv[1],"r");
  }
  yyparse();

  return 0;
}
```

To generate and execute this program the following commands have to be started:

```
> flex calc.l
> bison -d calc.y
> gcc calc.tab.c lex.yy.c -lfl -o calc
> ./calc [<input-file>]
```

2.2 Eli Basics

Eli is an expert system, which can be used for constructing standard programming languages, extensions for them and special purpose languages. It reduces the cost of producing a compiler, in a way, that only precise descriptions of problems and their solutions must be defined. The main concept is, that Eli splits the problem of defining and implementing a compiler into a number of smaller subproblems (see figure 2.2), which become easier to describe and solve. The solving process of these problems is defined by declarative specifications instead of writing algorithmic code. The Eli system takes these specifications and generates a language processor without user interaction. Most other compiler construction tools, like Lex, a scanner generator, and Yacc, a LALR(1) parser generator, are considered by many as too slow (Lex) or not providing adequate error recovery (Yacc). The Eli system is developed by one group and not like most of the other compiler construction tools independently of each other. The performance of an Eli generated compiler is, because of that, better than the result of most of the other compiler construction tools.

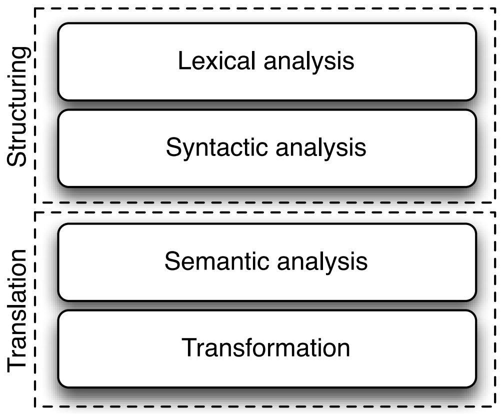


Figure 2.2: Subproblems, that have to be solved by the language developer

Not all of the subproblems have to be solved by hand again. Eli has a library of precoined solutions, which can be used by the language developer. In the Eli system solving process, redundancy from the specifications are removed, all of the processing that has to be done in order to let the different tools work together is hidden, and user request for products are simplified. It is also easy to extend Eli by adding tools or replace existing tools with a newer version.

Because Eli is a very complex system, I will show the main basic concepts of specifying a language processor.

2.2.1 Invoking Eli and Creating a Project

In the Eli system a project consists of a set of specifications, which are automatically recognized by the Eli system. To start Eli, the user should run "`eli`" in the directory, where the specification file is located, to invoke it. The Eli shell appears and the system can be controlled. With "`!`*`command`*" a Unix shell command can be executed. If the user wants to open or view a file, for example "*`xyz.gla`*", he writes the filename followed by the direction. "`>`" means output and "`<`" input. For defining a compiler, a `<project-name>.specs` file is created and all necessary files for specifying this compiler are written line by line in it (as shown in example 2.10).

Example 2.10 possible contents of a `.specs` project file

```
scanner.gla
grammar.con
[...]
```

2.2.2 Lexical Analysis

"The function of the lexical analyzer is to read the source program, one character at a time, and to translate it into a sequence of primitives units called tokens." [3]

For defining a lexical analyser, the Eli user has to create at least one "`.gla`" file, which holds definitions of tokens, which have the syntax "`Token: $`*`regular-expression`*". In each line, only one token can be defined and it is also possible to define comments, or other tokens that should not be returned by the lexical analyzer and simply skipped. Eli also provides predefined expressions, called "canned descriptions", which are token definitions from other programming languages like C and Fortran.

Example 2.11 `.gla` file with token definitions, canned descriptions and an auxiliary scanner

```
Identifier: C_IDENTIFIER
            C_COMMENT
String:     C_STRING_LIT
Integer:    PASCAL_INTEGER
            $-- (auxEOL)
Float:      $(([1..9][0..9]*)|(0)).(([0..9]*[1..9])|(0))
```

Example 2.11 shows a possible use of canned descriptions (`C_IDENTIFIER`, `C_COMMENT`, `C_STRING_LIT` and `PASCAL_INTEGER`). For more information of canned descriptions see the Eli manual [5, see token sets].

It is also possible to include user defined scanning routines in C/C++ code in the lexical analyzer. This hand written code is called “auxiliary scanner”. These scanners (like line 5 in example 2.11 “`(auxEOL)`”) are invoked after a precondition, in this case the recognized pattern “`--`”, is met. In this example, the auxiliary scanner reads all characters to the end of line (EOL).

An additional part of the scanner is the token processor. By specifying directives in brackets, such as “`[mkstr]`” or “`[mkidn]`”, after a token definition, the recognized token is stored into a string table or is added to an identifier table, as example 2.12 shows.

Example 2.12 adding token processors to tokens

```
Identifier: C_IDENTIFIER     [mkidn]
Float:      $(([1..9][0..9]*)|(0)).(([0..9]*[1..9])|(0))  [mkstr]
```

For more information see [5, Token Processors – Lexical Analysis].

2.2.3 Syntactic Analysis

In the syntactical analysis the purpose is to determine the structure of the input program. Grammar rules have to be specified in at least one“`.con`” file and this must be declared in the “`.spec`” file.

The Eli system uses the Backus-Naur-Form for describing a grammar, but can also recognize grammar in the Extended-Backus-Naur-Form (EBNF), which is automatically translated to BNF in an internal generating stage by an Eli tool called “maptool”.

Example 2.13 A grammar in Eli syntax in EBNF

```
1 Program:   [Constants &'handle_constants();'] [Variables] Body .
2 Body:      Statement* .
3 Statement: Type Identifier ['=' ArithmeticExpression] ';' .
```

Example 2.13 shows a grammar in Eli syntax. If a pattern is surrounded by brackets, it is optional and an asterisk after a (non-)terminal means a repetition of that symbol for zero or more times. A plus instead of the asterisk specifies at least one or more repetitions.

It is also possible to embed semantic records in Eli. The Eli user can write at the right hand side “`&'`*`commands`*`'`”. These semantic records are verbatim copied to the generated parser and executed when the left side symbol of this action has been parsed (see line 1 in example 2.13).

It is possible to obtain from the parser generator debug information about the generated parser. In this information it is included, whether the parser has parsing conflicts, is suitable for LALR(1) parsing and the parsing automaton with its parsing states is described. To obtain this information the derivation `:parsable` has to be derived from the specification file.

2.2.4 Semantic Analysis

The central concept of the semantic analysis in Eli is the computation in trees. The Eli system converts the concrete syntax tree from the parser to an abstract syntax tree. An abstract syntax consists like a concrete syntax of a context-free grammar. It is usually ambiguous and the translation phase is based on it. The parser actions specify the tree construction. Some chain productions only have use for the syntax and not for the meaning, therefore such symbols are mapped to one abstract symbol. However semantically relevant chain productions are kept. This abstract syntax tree is the central data structure for the semantic analysis. The concept for that is to add attributes and its computations to the program tree. The main tasks for semantic analysis are name analysis, properties of program entities, type analysis and operator identification. All this information can be computed with a tree walking algorithm that calls functions of semantic modules in specified contexts and in an admissible order (see [7, slide 416]).

Figure 2.3 shows a conversion process that is automatically done by the Eli system. An example tree for this grammar could be figure 2.4 for the sentence "`5+5*3`".

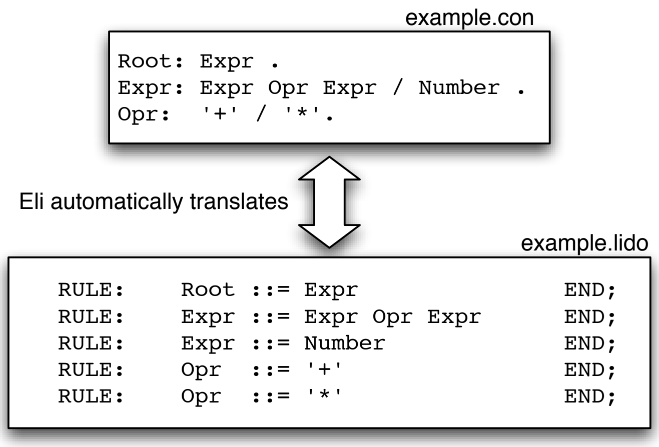
To each production rule, attributes can now be added. Example 2.14 shows possible computation for the rules. These rules are stored in a "`.lido`" specification file. Computations are denoted like calls of C functions and start with the keyword "`COMPUTE`" and end with "`END;`".

An attribute is defined as follows:

"`ATTR <attribute name> (: <type> ) ;`"

It is not possible to add attributes to literal terminals like `'+'` or `'*'` or access them. Terminals have per default `int` as their semantic type.

An attribute of any symbol that occurs in the rule context may be used in a computation of that context. The type of the attribute can be any C type. If the attribute has no type it is a state attribute, which has automatically the "`VOID`" type. These state attributes do not need to be defined by an "`ATTR`". Furthermore they are valid throughout the whole specification and can be overridden by



```
Root: Expr .
Expr: Expr Opr Expr / Number .
Opr:  '+' / '*'.
```

```
RULE:     Root ::= Expr                END;
RULE:     Expr ::= Expr Opr Expr       END;
RULE:     Expr ::= Number              END;
RULE:     Opr  ::= '+'                 END;
RULE:     Opr  ::= '*'                 END;
```

Figure 2.3: Automatic conversion from a grammar file to a LIDO specification and vice-versa done by Eli

attribute properties specified in “`SYMBOL`” constructs.

With the “`SYMBOL`” construct it is possible to bind an attribute only to a specific symbol. The syntax for that is the following:

“`SYMBOL <symbol name> : <attribute 1> , ... , <attribute n> : <type> ;`”

If an attribute is computed in the lower context it is called *synthesized* and if it is computed in the upper context it is called *inherited*. Any attribute must belong to either one of the classes in order to obey the single assignment rule.

If a symbol has more occurrence in a rule it can be accessed by adding the number of occurrence in brackets directly after the symbol name (as example 2.14, line 9-11, shows).

Sometimes it is necessary to wait for a special action until enough information is gathered. Pre- and postconditions can be used to solve that problem. The action uses attributes with no value which does not have to be defined. The postcondition is set if the action was executed and the action itself can only execute if the precondition is set. The syntax for that is as follows:

“`<postcondition> = <action> <- <precondition> ;`”

There are also three ways to gather information for a computations:

- access to a subtree root with “`INCLUDING`”

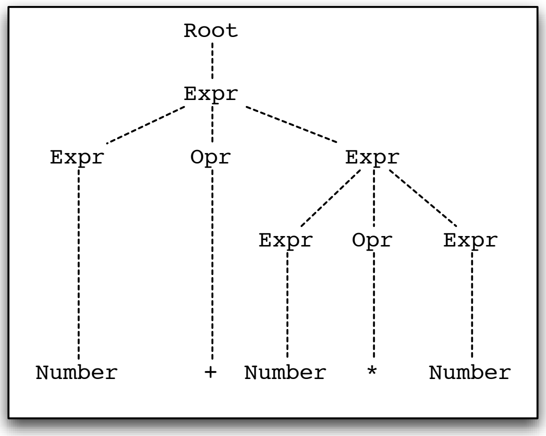


Figure 2.4: Example derivation tree for the grammar in figure 2.3

- access to context within a subtree from its root context with

```
CONSTITUENT(S) <object>
   [ WITH ( <type>, <combine function>, <single function>, <none function> ) ]
```

- access from left to right with "`CHAIN`"

For more detailed information of this computation functions see [5, LIDO - Computation in trees].

It is also possible to concatenate semantic actions to symbols only. The attributes are now accessed with "`SYNT`", for synthesized attributes, and with "`INH`" for inherited attributes (see example 2.15).

A complete documentation can be found in [5, LIDO - Computation in trees].

2.2.5 Transformation

The transformation from a tree to any kind of structured text can be done with the Eli tool PTG, the pattern based text generator. The structure of the output text is described by patterns in "`.ptg`" files. PTG generates functions, which can be used in the LIDO tree computation and may also be used in C functions or stand-alone C programs. The syntax of a PTG specification is as follows:
A pattern is specified by a named sequence of C string literals and $ tokens that denote insertions points (= arguments). C style comments are allowed. The generated function for a pattern has a prefix `PTG<functionName>`. The standard pattern argument type is `PTGNode`, which is also the return value of any PTG generated function. It is important that patterns must not have the same name and

Example 2.14 Example of an attributed grammar

```
ATTR value: int;

RULE: Expr ::= Number COMPUTE
  Expr.value = Number;
END;

RULE: Expr ::= Expr Opr Expr COMPUTE
  Expr[1].value = Opr.value;
  Opr.left   = Expr[2].value;
  Opr.right  = Expr[3].value;
END;

SYMBOL Opr: left, right: int;

RULE: Opr ::=  '+'  COMPUTE
  Opr.value  =  ADD(Opr.left, Opr.right);
END;

RULE: Opr ::=  '*'  COMPUTE
  Opr.value = MUL(Opr.left, Opr.right);
END;
```

Example 2.15 Synthesized attribute computation

```
ATTR Count: int;

SYMBOL Usage COMPUTE
  SYNT.Count = 1;
END;
```

must not collide with identifiers, that are predefined for PTG. PTG also does not insert any additional whitespace before or after elements of the pattern sequence and token separation and new lines have to be specified explicitly.
PTG support "indexed patterns", which means that arguments can be numbered and repeated in patterns (see example 2.16). It is now allowed to mix indexed and non-indexed arguments in a pattern.

There are some more PTG properties:

- **Typed Patterns:**
  It is also possible to give the arguments a special type (see grammar 2.1, production "*Type*"). If any typed argument occurs as an indexed argument more than one time in a pattern, it has to have the same type.

- **Function call insertions:**
  Functions can be called within pattern. For more information see [5, PTG -

PTGSpec	→	PatternSpec +
PatternSpec	→	PatternName `':'` (Item | Optional∗)
PatternName	→	`Identifier`
Item	→	`CString` | Insertion | FunctionCall
PatternName	→	`'$'` [`Number`] [Type]
FunctionCall	→	`'['` `Identifier` Arguments `']'`
Arguments	→	Insertion ∗
Type	→	`'int'` | `'string'` | `'pointer'` | `'long'` | `'short'` | `'char'` | `'float'` | `'double'`
Optional	→	`'{'` Item + `'}'`

Grammar 2.1: Syntax of PTG Specifications

Pattern Based Translation].

- **Optional parts in patterns:**
  Optional parts can be defined in curly brackets and they are only executed if all arguments are producing output.

There are three functions, which output the generated text:

1. `PTGNode PTGOut (PTGNode r);` – writes the information of the PTGNode `r` to the standard output `stdout`.
2. `PTGNode PTGOutFile (const char *f, PTGNode r);` – writes the information of the PTGNode `r` to a file `f`.
3. `PTGNode PTGOutFPtr (FILE *f, PTGNode r);` – writes the information of the PTGNode `r` to a stream `f`.

Example 2.16 Indexed PTG patterns

```
Link: "<a href=\"" $1 "\" alt=\"" $2 "\">" $1 "</a>"

------------------------------------------------------
PTGLink(
  PTGString("http://www.uni-paderborn.de"),
  PTGString("Universitaet Paderborn")
);
------------------------------------------------------
produces:

<a href="http://www.uni-paderborn.de" alt="Universitaet Paderborn">
  http://www.uni-paderborn.de</a>
```

2.3 GLR Basics

GLR parsing is a special parsing technique which was developed by Masaru Tomita (see [10]). Prof. Tomita developed the Generalized LR parsing technique (some call it "Tomitas algorithm") as a part of his Ph.D. thesis at Carnegie Mellon University. He tried to build some natural language systems based on existing parsing methods. However the existing methods were very slow, so that he tried to find another faster parsing method. He explained in his book [10, pp. 2f] a graph structured stack, which describes different stacks in GLR and described ambiguities in natural language. Also some improvements for his parsing method are described in the book. It is very difficult to parse natural language. For instance the sentence "I saw Jane and Jack hit the man with a telescope." has several meanings:

- `[I saw [[Jane and Jack] hit the man] with a telescope].`
- `[I saw [[Jane and Jack] hit the man  with a telescope]].`
- `[I saw Jane and [Jack hit the man  with a telescope]].`
- `[I saw Jane and [Jack hit the man]  with a telescope].`
- `[I saw [Jane and [Jack hit the man]  with a telescope]].`

Grammar 2.2 is a grammar which could process the input.

A complete parse trace for that sentence can be found in [10, pp. 7-14]. Figure 2.5 shows the complete parsing tree in the last parsing step. A more detailed analysis of Tomita's algorithm can be found at [9].

A GLR parser uses the same basic LALR(1) algorithm and the same LALR(1)

$$
\begin{array}{rcl}
\mathrm{S} & \rightarrow & \mathrm{NP\ VP} \\
\mathrm{S} & \rightarrow & \mathrm{S\ PP} \\
\mathrm{S} & \rightarrow & \mathrm{S\ and\ S} \\
\mathrm{NP} & \rightarrow & \mathrm{n} \\
\mathrm{NP} & \rightarrow & \mathrm{det\ n} \\
\mathrm{NP} & \rightarrow & \mathrm{NP\ PP} \\
\mathrm{NP} & \rightarrow & \mathrm{NP\ and\ NP} \\
\mathrm{VP} & \rightarrow & \mathrm{v\ NP} \\
\mathrm{VP} & \rightarrow & \mathrm{v\ S} \\
\mathrm{PP} & \rightarrow & \mathrm{p\ NP}
\end{array}
$$

Grammar 2.2: GRA: A non-LR grammar

tables as an ordinary LALR(1) parser. However the behavior changes. If the parser encounters a shift/reduce or a reduce/reduce conflict (for GLR in Bison see "parsing conflicts" in [6, pp. 72.-79]), it effectively splits the parser into multiple parsers, one for each possible shift or reduction. These parsers progress in lock-step, which means that they all consume one token from the scanner at a time and wait until each scanner consumed that token before they proceed. When such a split happens the parsing stack is duplicated and each stack represents now a guess as to what the proper parse is. When these parsers get additional input they can indicate that they are either wrong with their guess or right. If the guess was wrong, the stack is deleted and all semantic actions are also canceled. If it was right, the stack and with it the semantic actions are kept alive until the stacks are equal and can be merged or the input is processed.

Grammar 2.3 shows a grammar which has a reduce/reduce conflict. This grammar could be parsed by a LALR(2) parser. In this case, the grammar is not ambiguous and in Bison it would be enough to toggle to GLR mode without any changes in the grammar. This simple method is only applicable for unambiguous grammars. Otherwise it could be that not all stacks merge to one at the end and multiple semantic actions are executed. Bison has some special techniques to give stack precedence or handle by hand-written code how to merge stacks. Consider listing 2.3 as an example for an ambiguous grammar defined in a Bison specification.

Listing 2.3: glrexmpl.y - ambiguous grammar example for GLR

```
/* PROLOGUE */
/* DEFINITIONS */

```

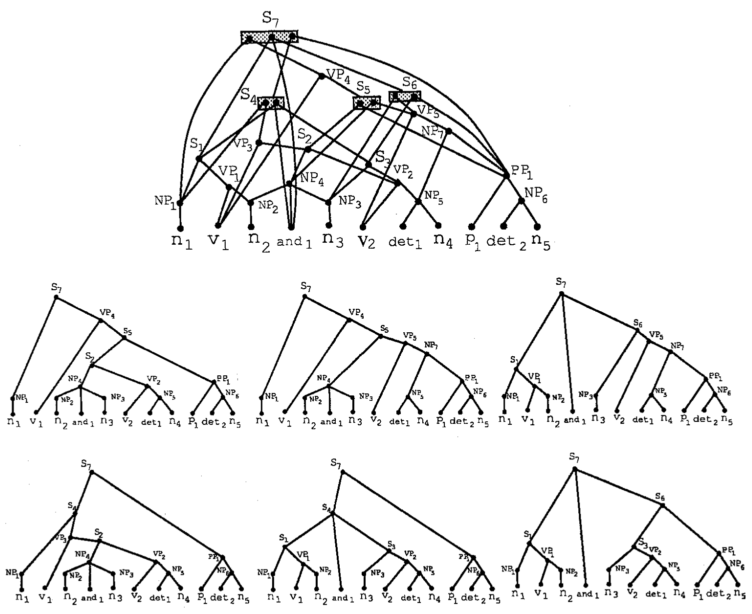


Figure 2.5: Final parsing tree for the sentence "I saw Jane and Jack hit the man with a telescope."

```
%glr-parser

%%
/* GRAMMAR */
prog   : /* empty */
       | prog stmt
       ;

stmt   : expr ';' %dprec 1
       | decl     %dprec 2
       ;

expr   : ID
       | TYPENAME '(' expr ')' { /* typecast */ }
       | expr '+' expr
       | expr '=' expr
       ;

decl   : TYPENAME declarator ';'
       | TYPENAME declarator '=' expr ';'
       ;

declarator : ID
           | '(' declarator ')'
           ;
```

$$
\begin{array}{rcl}
S & \rightarrow & (A) \\
  & | & B..B \\
A & \rightarrow & \mathbf{x} \\
  & | & A,\ \mathbf{x} \\
B & \rightarrow & \mathbf{x} \\
  & | & (B)
\end{array}
$$

Grammar 2.3: Unambigous grammar which is not in LALR(1) because there is a reduce/reduce conflict.

The sentence "`T (x) = y+z;`" parses either an `expr` or a `stmt`. Both parsing stacks are equal, but different semantic actions have to be invoked. By default Bison invokes both recorded semantic action sets, however it is possible to select a special one with `%dprec <n>`. This directive gives the precedence which parsing stack and semantic stack should be taken (`n` is the priority).

3 Design of a Bison Package for Eli

3.1 Defining an Eli Package

In this section I will describe how to define an Eli package. I will refer most the of the time to the Odin manual [4], in which all steps of defining a package for the expert system are described in detail.

3.1.1 Tool Packages

Any package in the Odin system can hold information about object types and parameter types and each is specified in special directories called in the Odin manual *tool package libraries*. When a new package is created it is important to rebuild the cache with "`eli -R`" in order to get the new package available. Any new package has to be defined in the `PKGLIST` file, which is located in the root directory of the packages. The default location for the packages is `/usr/local/lib/Eli/pkg`.

A new defined tool package, for example tool `abc` has to contain a *derivation graph* (see [4]) with the same name and the extension `.dg`. In this case the folder `abc` has to contain a file `abc.dg`.

It is also possible to add version numbers to a package. The package can contain a file `LIBVER` which specifies the version. For more information see [4, p. 26f].

3.1.2 Derivation Graphs

In a derivation graph source declarations, object types, parameter types, environment variables and tool declarations can with a special language be specified.

Source Declarations

In a source declaration a special kind of a source file is assigned to an object. For example:

```
*.bison => :bisonSpec;
*.y     => :parserGrammar;
*       => :FILE;
```

Object Type Declaration

The specified objects have to be declared. Any object has to be derived to a special built-In supertype [4, p. 28ff]. To declare an object the objectname with an identifier, a right arrow and the direct supertype have to be specified. For example:

```
:bisonSpec 'Bison specification file' => :FILE;
```

Parameter Type Declarations

A parameter type can be declared in the following way:

```
+specialParam 'A special parameter' => :first;
```

For more information see [4, p. 32]

Environment Variable Declaration

Environment variables can be declared as an environment variable in a UNIX shell. For example:

```
$BISON 'The bison parser generator' = '/usr/bin/bison'
```

These environment variables are only defined in a special Eli cache.

Tool declaration

The tool declaration consists of the action, the parameters for this tool and the output of this tool. For example:

```
EXEC (/bin/yacc) -dv (:y) => (:y.tab.c) (:y.tab.h) (:y.output);
COLLECT (:bisonSpec) (:bisonSpec :map=:all_bisonSpec) => (:all_bisonSpec);
  :all_bisonSpec 'All bison specificatoins' => :LIST;
READ-LIST (:bisonSpec) => (:bisonSpecList);
READ-REFERENCE (:parserGrammar) => (:parserGrammar.ref);
```

This declares three output objects. If any of these objects is called the command `/bin/yacc` is executed with the given parameters.

There are several tools, which can be invoked:

- `EXEC` Tool: This tool executes a command.
- `COLLECT` Tool: There are two different output types the `COLLECT` can have.
  - List: If the output is a list, the `COLLECT` tool produces a list whose elements are the input objects.
  - Reference: If the output is a reference, the `COLLECT` tool produces a reference to the input object.

- `READ-LIST` Tool: This tool produces a list from an inpt object.
- `READ-REFERENCE` Tool: This tool produces a reference to the object namend by the single odin-/eli-expression in the input file.

3.2 Integrating Bison in Eli

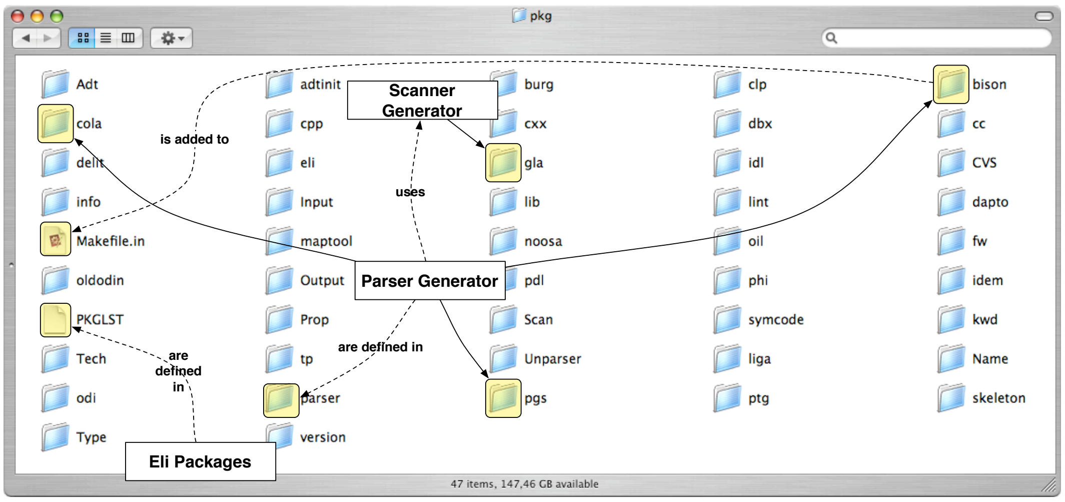


Figure 3.1: All Eli packages, their dependencies and changes made in order to integrate Bison

There are several steps which have to be performend in order to integrate Bison as a new tool package in Eli.

Figure 3.1 shows that Eli contains a lot of packages. This chapter covers the extraction of important packages in order to add Bison as a new package to Eli. Furthermore it highlights these adjusted packages and shows their dependencies.

3.2.1 Obtaining the Source

I obtained the latest CVS source code from the Eli version 4.4.3 from the public Sourceforge CVS server. The source also includes an Odin version, which I replaced by the latest stable Odin version, which is included in the latest stable Eli version which is also downloadable from the Eli Sourceforge project page. I did this, because it was not possible for me to compile the CVS Odin source.

The building process of the source is managed by autoconf, which generates configure scripts.

3.2.2 Adding a New "bison" Package to Eli

Like all Eli tools, a new tool has to be in a new folder in the Eli packages directory `pkg` and the name of it must also be written in the file `PKGLST` to tell the Eli system, that there is another package.

The new Bison package contains the following files:

- `bison.dg` – The tool derivation graph file
  In this file three new derivations are defined:
  - `:bisonGen` – Generation of the concrete parser
  - `:bisonInfo` – Information of the generated parser
  - `:bisonSpecification` – Bison specification file
- `bison.h` – The scanner interface header file
- `pgram2bisonGen.sh`
  This shell script invokes the "pgram2bison" translator and translates the "pgram" style grammar into a Bison specification file and provides the derivation `bisonSpecification`.

```
ODIN_pgram=$1;shift; ODIN_bupgram=$1;shift;

"$ODINCACHE/PKGS/bison/pgram2bison.exe" "$ODIN_pgram" >> \
bisonSpecification
```

- `bisonInvoker.sh`
  This shell script invokes Bison and stores the header file `bison.h` and the generated GLR-parser `bison.c` into a derived directory which is provided in the derivation `:bisonGen`. Also debug information from Bison is stored in the derivation `:bisonInfo`. It gets as input a Bison specification file.

```
ODIN_spec=$1;shift;BISON_opt=$1;shift;BISON_e128=$1;shift;

echo "==================================" >> bisonInfo
/usr/bin/bison -v "$ODIN_spec" -o bison.c 1> bisonInfo 2>&1
mkdir bisonGen
cp "$ODINCACHE/PKGS/bison/bison.h" bisonGen
if test -s bison.c
then
  cp bison.c bisonGen
else
  echo "Generated parser not found. There must be an error occured \
        during the generation process." >> ERROR
  exit 0
fi

if test -s bison.output
then
```

```
  echo "====================================" >> bisonInfo
  cat bison.output >> bisonInfo
  echo "====================================" >> bisonInfo
fi
```

- `version` – The version number of the tool
- `../src/` – The source folder of the “pgram2bison” translator. The source is obtained by deriving `:source` from the `pgram2bison.specs`.
- `Makefile.in` – The makefile for this package
  This makefile specifies which files have to be copied to the final product and which programs have to be build first.

```
CC  = @CC@

DIR = pkg/bison
SRC = bison.dg bison.h
AUX = Makefile.in Makefile version
CMD = bisonInvoker.sh pgram2bisonGen.sh
EXE = pgram2bison.exe
SRCDIR  =
AUXDIR  = src

@toolmk_h@

pgram2bison.exe:  src/pgram2bison.exe
  @rm -f $@
  @LINK@ $? $@
```

3.2.3 Changing the “parser” Package

The “parser” package in the Eli tool package is responsible for the different parsers. A third parser, Bison, has to be added to the scripts. Therefore some files have to be changed:

- `parser.dg` – The derivation graph file
  The possible selection of Bison as a parser in the derivation graph file has to be implemented. Therefore two tool declarations have to be adjusted.

```
EXEC (parserOut.sh) (.) (:LIST :extract=:mapPgram :names)
                    (:pgsGen :name) (:colaBe :name)
                    (:bisonGen :name) (+parser) (+monitor)
  => (:parserOut);
:parserOut 'Results of parser generation (PGS, COLA or BISON)' => :FILE;

EXEC (infoOut.sh) (:LIST :extract=:mapPgram :names)
      (:colaInfo :name) (:pgsInfo :name)
      (:bisonInfo :name) (+parser)
  => (:infoOut);
```

Here appears now the derivation from the new Bison package `:bisonGen` and `:bisonInfo`.

- `parserOut.sh.in`
  This shell script handles the possibles parsers. It evaluates the `+parser` switch and selects regarding to that the correct parser generator. By default the parser generator is PGS.

```
PKG=$1;shift; ODIN_e115=$1;shift; ODIN_pgs=$1;shift;ODIN_cola=$1;
shift;ODIN_bison=$1;shift;ODIN_parser=$1;shift;ODIN_mon=$1;shift;

if test -s "$ODIN_e115"
then
  echo "$PKG/parser.reqmod" > parserOut
if @SHMONITOR@
then
  if test '' != "$ODIN_mon"
  then
    echo "$PKG/mon_cprods.c" >> parserOut
  fi
fi
  if test '' = "$ODIN_parser" -o 'pgs' = "$ODIN_parser"
  then cat "$ODIN_pgs" >> parserOut
  elif test 'cola' = "$ODIN_parser"
  then cat "$ODIN_cola" >> parserOut
  elif test 'bison' = "$ODIN_parser"
  then cat "$ODIN_bison" >> parserOut
  elif test 'none' = "$ODIN_parser"
  then cp /dev/null parserOut
  else
    echo "\"$ODIN_parser\" is not a valid parser generator name" > /
    ERRORS
    exit 0
  fi
else echo "$PKG/dfltparse.reqmod" > parserOut
fi
```

- `infoOut.sh`
  This script selects the debug information from the parser generator, such as the parser states and/or parsing conflicts.

```
if test -s "$ODIN_e119"
then
  if test '' = "$ODIN_parser" -o 'pgs' = "$ODIN_parser"
  then cp "$ODIN_pgs" infoOut
  elif test ' ' = "$ODIN_parser" -o 'cola' = "$ODIN_parser"
  then cp "$ODIN_cola" infoOut
  elif test 'bison' = "$ODIN_parser"
  then cp "$ODIN_bison" infoOut
  else
    echo "\"$ODIN_parser\" is not a valid parser generator name" > /
    ERRORS
    cp /dev/null infoOut
  fi
else
  echo "There is no grammar to be tested for parsability" > ERRORS
  cp /dev/null infoOut
fi
```

3.2.4 Adjusting Building Files

In order to get the new package working the following files have also to be adjusted:

- `Eli/pkg/Makefile.in`
  This Makefile contains a list of all packages, which should be generated. Bison must be added to that list.

```
CC  = @CC@

DIR = pkg
SRC =
AUX = Makefile Makefile.in
CMD =
EXE =
GEN =
GENINS  =
SRCDIR  = dapto Adt Input Name Output Prop Scan Tech Type Unparser \
    adtinit burg cc clp cola cpp cxx dbx delit \
    eli gla idl kwd lib liga lint \
    maptool noosa odi oldodin parser pdl pgs bison phi ptg \
    run skeleton symcode version \
    fw info oil idem tp
AUXDIR  =

@toolmk_h@
```

- `Eli/pkg/PKGLST` – Package name file
  This file holds the names of all available packages in the Eli system. Because the names of the packages are line separated, a new line with “bison” is added.

- `Eli/configure.in` – Main configure file for Eli
  In this file I added a script, which checks whether the user has a valid Bison version installed:

```
dnl ##############################################################
dnl -- Check wether bison is installed

AC_PATCH_PROG([BISON],[bison],[nobison])
if test $BISON == "nobison" then
  AC_MSG_ERROR([No Bison found, please install the GNU Bison
  parser generator. For more information see:
  http://www.gnu.org/software/bison/])
fi
```

  Furthermore the Bison makefile was added in the `AC_OUTPUT` script.

3.2.5 Invoking `autoconf`

All necessary changes for adding a Bison package to Eli are now accomplished and `autoconf` has to be run in this order to generate correct `configure` script in these folders:

- `Eli/pkg/eli/ofiles/`
- `Eli/pkg/cpp/gnu_cpp/`
- `Eli/pkg/skeleton/gnu_sed/`
- `Eli/`

It is important that `autoconf` knows the `aclocal.m4` in the Eli root directory.

3.3 Grammar of the Eli Parser Grammar "`pgram`"

Grammar 3.1 shows the reverse engineered grammar for the pgram format. Because there was no concrete definition in the Eli manual, I analyzed several grammars and confered with my supervisor about the correctness of my pgram grammar.

Here are some explanations of the terminal symbols and their usage:

- `symb` – A symbol can represent either a terminal symbol or a nonterminal symbol and is formed like any C-like string.
- `litsymb` – A literal symbol represents a nonterminal symbol which can hold any characters except the single quotes, because it is surrounded by them.
- `semanticEntry` – A semantic entry starts with `&'` and ends with a single quote. Within it the terminal can contain a list of C statements.
- `number` – The number terminal holds a positive integer number

Grammar	→	Specifications StartSymbol Productions*
Specifications	→	`'$CODE'` TokenDefinitions* `'$SEPA'`
TokenDefinitions	→	`symb '=' number '.'` | `litsymb '=' number '.'`
StartSymbol	→	`STARTSYM symb '.'` | ϵ
Productions	→	`symb ':'` Sequence `'.'`
Sequence	→	Symbols* `'/'` Sequence | Symbols*
Symbols	→	`symb` | `litsymb` | `semanticEntry`

Grammar 3.1: Pgram Grammar

3.4 Using Gla Generated Scanners with Bison

Gla is the scanner generator in the Eli system. It uses Lex to generate the parsing tables and uses its own algorithm to process them. The header file `gsdescr.h` (gsdescr stands for "Grundsymboldescriptor" [ger.] base symbol descriptor) holds definitions of the base symbol types used by the scanner. I changed the names of these types:

```
/*** Types ***/
#define TOKENTYPE GRUNDSYMBOLDESKRIPTOR
#define CODETYPE    TERMINALSYMBOL
#define ATTRTYPE    ATTRTYPE
```

I also used a macro which lets me access the token code, token position, token line number and token attribute of a given token `tok`:

```
/*** TOKENTYPE-Macros ***/
#define CODE(tok) (T_CODE((*tok)))
#define POS(tok)  (T_POS((*tok)))
#define LINE(tok) (LineOf(T_POS((*tok))))
#define ATTR(tok) (T_ATTR((*tok)))
```

Then all that is left to do is to define the function, which returns the next token (`TOKEN(tok)` and a token variable which holds the actual token (`Token`):

```
#define TOKEN(tok)  GET_TOKEN(*(tok))
static  TOKENTYPE token[1];
TOKENTYPE *Token = token;
```

This is the interface for Gla and it is combined in the `bison.h` header file, which is automatically included in any translator which uses Bison generated parsers.

yylex - The Bison Scanner Function

As already described in the Bison introduction, Bison needs to call a special function to obtain a new token, called `yylex`. This function is generated with the Bison specification by the program pgram2bison and is also included in any Bison generated paser.

```
1  int yylex(void) {
2     TOKEN(Token);
3     yylval = *Token;
4     int retval =  T_CODE(yylval);
5     if (retval == 1) return -1;
6     return retval;
7  }
```

The TOKEN(Token); statement is the Gla scanner function. The scanned token is saved in Token. It is also needed to store the token in a special variable called yyval to refer in the semantic analysis to this token.
Gla uses a special token for EOF, but Bison has to get zero or a negative value for recognizing an end of file. This is corrected in line 5 of the above example.

4 Implementation

The parser generators PGS and COLA, which are in the Eli system, need the parser grammar in a special format, which is called pgram and already specified in the previous chapter. The Eli user defines the grammar in a `.con` grammar file. This possible EBNF input is taken by a tool called maptool and translated into a BNF form. Furthermore semantic actions for parse tree building are added to each production.

To build an interface between the pgram format and the bison format, a translator has to be defined, which converts pgram syntax grammar into a bison specification file (see figure 4.1).

In this chapter I will show the necessary steps for defining such a translator in detail.

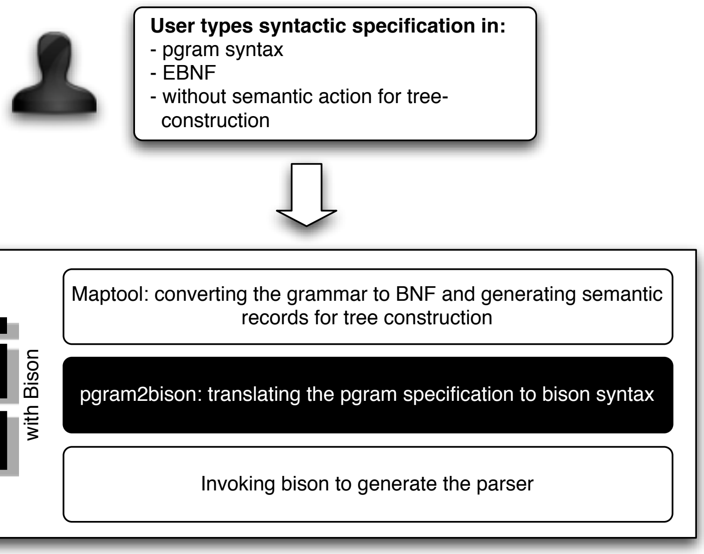

User types syntactic specification in:
- pgram syntax
- EBNF
- without semantic action for tree-construction
ELI
with Bison
Maptool: converting the grammar to BNF and generating semantic records for tree construction
pgram2bison: translating the pgram specification to bison syntax
Invoking bison to generate the parser

Figure 4.1: Transformation process in Eli

```
symb:       C_IDENTIFIER          [mkidn]
litsymb:    $[’] (SpecSemantic)   [mkCustomStr]
semantic:   $[&][’] (SpecSemantic)[mkCustomStr1]
number:     $[0-9]+               [mkidn]
            $\$[A-Z]+
            $\$ (skip)
            C_COMMENT
```

Listing 4.1: word.gla - lexical specification of pgram2bison

4.1 Generating a Translator with Eli

4.1.1 Scanner Definition

There are four different tokens the parser needs:

- Symbol: A symbol looks like a C-like identifier and represents either a terminal or a non-terminal
- Literal Symbol: A literal symbol represents a terminal. Any symbols between single quotes are allowed.
- Semantic Record: A semantic records holds the complete semantic record. In front of it, there must be a `&’` and afterwards a single quote.
- Number: An integer number.

Example 4.1 Dangling else shift/reduce conflict resolve in pgram

```
IfStatement: ’if’ Expression ’then’ Statement $’else’ &’[..]’ .
IfStatement: ’if’ Expression ’then’ Statement ’else’ Statement &’[..]’ .
```

As it is shown in listing 4.1 some strings must be skipped. These are any strings which start with a dollar sign. These tokens are unused items in the grammar and are not needed by Bison. Also C-like comments shall be allowed. There is another token which can appear in a pgram specification. To resolve for shift reduce conflicts a dollar sign can be inserted to tell the parser that if the next token is the token behind the dollar sign, these special productions should not be executed and another production with extends the original production should be used (see example 4.1). In Bison this is not possible, because it has another way to resolve such conflicts. Therefore an auxiliary scanner skips all characters until one of the following characters appear:

- Slash (`/`) – Another production starts
- Dot (`.`) – The production ends

- And sign (&) – A semantic action starts

The auxiliary scanner `SpecSemantic` reads all characters until a single quote appears and the actions `mkCustomStr` and `mkCustomStr1` build from the scanned token different substrings. The first action cuts the first and the last character, the second action cuts the first two and the last characters. Both actions store the plain symbol or semantic action in the stringtable, because they are not needed for either semantic analysis or translation.

4.1.2 Syntax Definition

The definition of the syntax is based on the pgram grammar syntax from the previous chapter and only changed slightly for translation purposes.

```
Grammar:             Specifications StartSymbol Productions .
StartSymbol:         '<$START_SYMBOL>:' StartSymbolSymbol '.' / .
StartSymbolSymbol:   symb .
Specifications:      '$CODE' TokenDefinitionList '$SEPA' /   .
TokenDefinitionList:TokenDefinitions .
TokenDefinitions:    TokenDefinitions TokenDefinition
                     / TokenDefinition .
TokenDefinition:     symb '=' number '.' / litsymb '=' number '.' .
Productions:         ProductionsList .
ProductionsList:     ProductionsList Production / Production .
Production:          LHS_symb ':' SequenceList '.' .
SequenceList:        SequenceList '/' SequenceEntry / SequenceEntry.
SequenceEntry:       Sequence .
Sequence:            Sequence Symbol  / .
Symbol:              symb / litsymb / SemSymbol .
SemSymbol:           semantic .
LHS_symb:            symb .
```

4.1.3 Attribute Analysis

In the attribute analysis it is important to change the semantic record regarding the Bison specifications while it is parsed. In Bison any semantic value is stored in the tree elements and referred by `$n` in the semantic actions, where `n` is the number of symbol on the right hand side starting with one (see Bison Basics). However PGS and COLA do not have this feature and use a tokenstack, where all possible tokens are located and referred by the macro `TokenStack(n)`. A mapping function called `action(int semantic)` was written, where the semantic value is transformed to Bison value.

$$\texttt{TokenStack(n)} \rightarrow \texttt{\$(n+1)}$$

The semantic computations are defined in the file `SemanticRecordChange.fw`:

```
@p maximum_input_line_length = infinity
@O@<symbol.lido@>@{
```

```
ATTR Sym: int;

RULE: SemSymbol ::= semantic COMPUTE
  SemSymbol.ready = action(semantic);
END;

RULE: Grammar ::= Specifications StartSymbol Productions COMPUTE
   Grammar.ready = CONSTITUENTS SemSymbol.ready;
END;
SYMBOL Entity INHERITS IdDefScope COMPUTE
  SYNT.Sym=TERM;
END;

SYMBOL StartSymbolSymbol INHERITS Entity END;
SYMBOL Semrec INHERITS Entity END;
SYMBOL Number INHERITS Entity END;
SYMBOL LHS_symb INHERITS Entity END;
@}
@O@<symbol.HEAD.phi@>@{
void action(int val);
@}
```

Because the input from pgram had already semantic checking, no further semantic checking for this translator is necessary.

4.1.4 Code Generation

The standard way of generating code in Eli is the use of patterns. For generating the translation patterns the following PTG Patterns were used:

- ***Grammar*** – The main Bison specification
  This is the main pattern, which describes the body of the Bison specification file. It includes the scanner and tree generating interface `bison.h` and defines the function `yylex` and the error function `yyerror`.
  The pattern function needs two parameters:

  - *Token definitions* – the definitions of the tokens, which are returned by the parser
  - *Grammar* – the grammar itself in Bison syntax

  Every parser needs to provide the parsing function `void Zerteiler (void)`, which is also implemented and returns the genuine parsing function provided by Bison `void yyparse (void)`.

```
Grammar:
  "/*******************************************/\n"
  "/*   Automated translated pgram bison      */\n"
  "/*                                         */\n"
  "/*       S P E C I F I C A T I O N         */\n"
  "/*                                         */\n"
  "/*  pgram2bison.exe (Ulf Schwekendiek)     */\n"
"/*                    sulf(at)upb.de         */\n"
"/*  Universitaet Paderborn 2007             */\n"
"/******************************************/\n"
"\n\n"
"%{\n"
"#include \"bison.h\"\n"
"#define YYSTYPE TOKENTYPE\n"
"%}\n\n"
"%glr-parser\n\n"
$ /* TokenDefinitons */"\n"
"%start " $ /* Start Symbol */"\n\n"
"%%\n"
$ /* Grammar */"\n"
"%%\n\n"
"int yylex(void) {\n"
"\t\n"
"\tTOKEN(Token);\n"
"\tyylval = *Token;\n"
"\tint retval =  T_CODE(yylval);\n"
"\tif (retval == 1) return -1;\n"
"\treturn retval;\n"
"}\n\n"
"void yyerror (char const *s) { printf(\"parse error: %s\",s); }\n\n"
"void Zerteiler (void) { yyparse(); }\n"
```

- ***Combine*** – This pattern takes two arguments and combines them.

```
Combine:
 $ $
```

- ***List*** – The list pattern takes two arguments and combines them separated by a space. It is used to separate sequence entries.

```
List:
  $ " " $
```

- ***ProductionSep*** – This pattern separates two productions with two new line commands.

```
ProductionSep:
  $ "\n\n" $
```

- ***Production*** – The production pattern translates a production in the Bison format.

```
Production:
  $ "\t:\t" $ "\n\t;\n"
```

- ***Single*** – This pattern represents a single argument.

```
Single:
  $
```

- ***SequenceSep*** – This pattern separates two sequences.

  ```
  SequenceSep:
    $ "\n\t|\t" $
  ```

- ***Tokendefintions*** – This pattern puts a new line after all token definitions.

  ```
  Tokendefintions:
    $ "\n"
  ```

- ***TDefInt*** – This pattern translates a token definition.

  ```
  TDefInt:
    "%token " $ " " $ "\n"
  ```

- ***TDefIntSpec*** – This pattern translates a special token definition. This is a token in double quotes (for more information see Chapter 1 - Introduction to Bison).

  ```
  TDefIntSpec:
    "%token TERM" $ " " $ " \"" $ "\"\n"
  ```

- ***Quoted*** – This pattern surrounds the argument with double quotations.

  ```
  Quoted:
    "\"" $ "\""
  ```

- ***Semantic*** – This pattern translates a semantic entry.

  ```
  Semantic:
    " \n\t{\n\t\t " $ " \n\t}\n\t\t"
  ```

- ***String*** – This pattern needs as argument a string value.

  ```
  String:       $ string
  ```

- ***Int*** – This pattern needs as argument an integer value.

  ```
  Int:          $ int
  ```

How the parsed grammar is translated is covered by the file `code.lido`.

```
ATTR Ptg: PTGNode;
SYMBOL Entity INHERITS IdPtg END;

SYMBOL Grammar COMPUTE
   PTGOut (PTGGrammar (
      CONSTITUENT Specifications.Ptg,
      CONSTITUENT StartSymbol.Ptg,
      CONSTITUENT Productions.Ptg));
END;

SYMBOL SequenceEntry COMPUTE
```

```
  SYNT.Ptg = PTGSingle(
    CONSTITUENTS Symbol.Ptg WITH (PTGNode, PTGList, PTGSingle, PTGNull)
    );
END;

SYMBOL Productions COMPUTE
   SYNT.Ptg = PTGSingle(
     CONSTITUENTS Production.Ptg
       WITH (PTGNode, PTGProductionSep, PTGSingle, PTGNull));
END;

SYMBOL Production COMPUTE
  SYNT.Ptg = PTGProduction(
     CONSTITUENT LHS_symb.Ptg,
     CONSTITUENTS SequenceEntry.Ptg
       WITH (PTGNode, PTGSequenceSep, PTGSequenceSingle, PTGNull)
  );
END;

SYMBOL TokenDefinitionList COMPUTE
   SYNT.Ptg = PTGTokendefintions(
      CONSTITUENTS TokenDefinition.Ptg
         WITH (PTGNode, PTGCombine, PTGSingle, PTGNull)
    );
END;

SYMBOL Specifications COMPUTE
   SYNT.Ptg = PTGSingle ( CONSTITUENTS TokenDefinitionList.Ptg WITH
    (PTGNode, PTGCombine, PTGSingle, PTGNull));
END;

SYMBOL StartSymbol COMPUTE
   SYNT.Ptg = PTGSingle (
     CONSTITUENTS StartSymbolSymbol.Ptg WITH
       (PTGNode, PTGCombine, PTGSingle, PTGNull)
   );
END;

RULE: Symbol ::= symb COMPUTE
   Symbol.Ptg = PTGString(StringTable(symb));
END;

RULE: Symbol ::= litsymb COMPUTE
   Symbol.Ptg = PTGQuoted(PTGString(StringTable(litsymb)));
END;

RULE: Symbol ::= SemSymbol COMPUTE
   Symbol.Ptg = SemSymbol.Ptg;
END;

RULE: SemSymbol ::= semantic COMPUTE
   SemSymbol.Ptg = PTGSemantic(PTGString(StringTable(semantic)));
END;

```

```
RULE: TokenDefinition ::=  symb '=' number '.' COMPUTE
   TokenDefinition.Ptg = PTGTDefInt (
        PTGString(StringTable(symb)),
        PTGString(StringTable(number))
   );
END;

RULE: TokenDefinition ::=  litsymb '=' number '.' COMPUTE
   TokenDefinition.Ptg = PTGTDefIntSpec (
        PTGString(StringTable(number)),
    PTGString(StringTable(number)),
    PTGString(StringTable(litsymb))
   );
END;
```

4.1.5 Generating the Executable and Obtaining the Source

In order to generate the executable for this specified translator, the `:exe` derivation has to be derived from the specification file and written to the disk drive with:

```
pgram2bison.spec :exe > pgram2bison.exe
```

The source can be obtained by deriving the `:source` derivation from the specification file. The output is written in a `src` folder:

```
pgram2bison.spec :source > src/
```

5 Evaluation

5.1 Installation Test

This test installs the new Eli installation on different operating systems, such as different Linux distributions, Apples OS-X and Microsoft Windows. It should be checked, whether the installation checks for a Bison executable and if not gives the user a warning message.

5.1.1 Ubuntu Linux 7.04 - Feisty Fawn

To install the Eli system under Ubuntu Linux, a basic Ubuntu installation needs two packages:

- libncurses5-dev
- bison

The first test for this operating system is an installation without the Bison package installed. The installation reports correctly, that it did not find any Bison executable and therefore the installation has to be stopped.

After installing the Bison package, the installation was successful.

5.1.2 Apple OS-X 10.4.10 - Intel/PowerPc

In order to install the Eli system under Apple's OS-X the developer tools from the Apple Installation DVD have to be installed. After that, no additional packages are required. The Eli installation was successful. However Bison does not work out of the box, because the version 1.28, which is installed by OS-X, does not support GLR parsing. After an update to version 2.3 Bison works properly. More information for this problem is in the next chapter (see "Known Problems").

5.1.3 Microsoft Windows XP

Under the Microsft Windows XP operating system, it is necessary to install Cygwin [2]. The full installation has all required latest versions of tools, including Bison, which are important for an Eli installation. After that, Eli was successfully installed and works properly.

5.2 Bison Parser with a Huge Grammar

To test the Bison generated parser I used the "Pascal-" specification. I changed to `Odinfile` and added `+parser=bison` and "Pascal-" was built correctly. After that I executed the test files, which all result in a correct solution.

5.3 GLR-Test

To test the GLR parsing method with the defined interfaces to the scanner and attribute evaluator I used a simple translator:

5.3.1 Test Translator

Lexical Analysis

The scanner accepts two different tokens:

- `ID` – A C-like identifier
- `NUM` – A positive integer number

The scanner also skips C-like comments.

`glr.gla`:

```
ID: C_IDENTIFIER [mkidn]
NUM: $[0-9]+
  C_COMMENT
```

Syntactical Analysis

The parse accepts strings for the following grammar:

```
start:  typedecl* .
typedecl: 'type' ID '=' type ';' .
type    :  '(' id_list ')' / expr '..' expr .
id_list : id_list ',' NUM / NUM .
expr    : expr '+' NUM / expr '-' NUM / NUM  / '(' NUM ')'.
```

This grammar has a shift/reduce conflict for the type production. The following strings should be accepted by the grammar:

```
type A = (1);
type B = (1)..(3-1);
```

However a LALR(1) parser can not resolve this problem very easily.

Semantic Specification

Because I only want to test the parser, I skipped the translation phase and only print some debug messages for the parsed productions:

```
RULE: typedecl ::= 'type' ID '=' type ';' COMPUTE
  printf("typedecl parsed\n");
END;

RULE: type ::=  '(' id_list ')' COMPUTE
  printf("Rule: type ::=  '(' id_list ')'\n");
END;

RULE: type ::= expr '..' expr COMPUTE
  printf("Rule: type ::= expr '..' expr\n");
END;
```

5.3.2 Test Input

The test input is as simple as the grammar:

```
type t = (5) .. 5;
type u = (5);
```

5.3.3 Test Results

The three different parsers are working properly.

- **PGS** – PGS reports the parsing conflict and is not able to process the input

```
Eli Version 4.4.3 (? for help, ^D to exit) (local: type ? for help)
-> glr.specs +parser=pgs :parsable >
PrintShift can't find a path
PGS 8.0 --- Input

Input Summary
=============
[...]

#
PGS 8.0 --- Analysis
Algorithm without elimination of chain productions
Algorithm without shift/reduce optimization

Analysis Results
================

Type                         : not LALR(1)
```

```
Number of errors            : 1
Conflict states             :   13
States                      : 26
Nonterminals                : 7
LR(0) reduce states         : 0

Conflicting Derivations
=======================

************************************************************************
   *** shift-reduce conflict on: ')'

start EOF
G1
G1 typedecl
   'type' ID '=' type ';'
                 '(' id_list ')'
                     |
                     NUM .  [REDUCE] id_list -> NUM  {')'} ?

-> input +cmd=(glr.specs +parser=pgs :exe) :stdout >
...
glr.specs.116043.Pgram 0 0 Grammar is not LALR(1)

->
```

- **COLA** – Cola also recognizes the shift/reduce conflict, responses with a debug information and is not able to process the input.

```
-> glr.specs +parser=cola :parsable >
**************************************************
** G R A M M A R   I S   N O T   L A L R (1) !! **
**************************************************

*************************
** S T A T I S T I C S **
*************************
Grammar                  : No Name
Type                     : IS NOT LALR(1)
Productions              : 13
Nonterminals             : 7
Terminals                : 12
States                   : 18
Successfull Modifications: NO
```

```
***************************
** P R O D U C T I O N S **
***************************
[...]

*******************
** S Y M B O L S **
*******************
[...]

*******************************************
** L A L R ( 1 ) - S T A T E - T A B L E **
*******************************************

CONFLICT-STATES:   10

################################################################################
STATE   10:     On Error: ,
--------------------------------------------------------------------------------
CONFLICTS:
    shift-reduce conflict (31) with set:   )
--------------------------------------------------------------------------------
 * |  R |  31:     id_list -> 'NUM' .     ERC:   )  ,
 * | ST |  22:        expr -> '(' 'NUM' . ')'
--------------------------------------------------------------------------------
(),SR:22)
################################################################################
-> input +cmd=(glr.specs +parser=cola :exe) :stdout >
[...]
ERROR: grammar is not LALR(1) !
```

- **Bison** – Bison detects also this parsing conflict, but because of the GLR flag it generates a parser although the grammar is not LALR(1). Also the generated translator is able to process the input correctly.

```
-> input +cmd=(glr.specs +parser=bison :exe) :stdout >
typedecl parsed
Rule: type ::= expr '..' expr
typedecl parsed
Rule: type ::=  '(' id_list ')'
->
```

5.4 Speed Test

Another important criteria for a parser is the parsing speed. A main aspect for integrating and using Bison instead of Pgs and Cola is that a Bison generated parser has no complexity differences. Therefore a test case was developed that measures the parsing and executing speed of an Eli generated translator with the three different generated parsers.

5.4.1 Test Grammar

As a test grammar I used the Eli example grammar "sets", which is already included in the Eli system and can be obtained in the following way:

- Invoke Eli and type ?, followed by a carriage return. This request will start a documentation browsing session
- Use the browser's goto command to place yourself at node `(novice)tskex`
- Use the documentation browser's `run` command to obtain a copy of the complete specification

The test is made up of two test files, which are processed by the sets translators, which hold each parser once. These test files are 5 MB and 20 MB and are executed 10 times.

Here is the ruby script which runs the test:

```
#!/usr/bin/env ruby
#
#  Created by Ulf Schwekendiek on 2007-07-20.
#  Copyright (c) 2007. All rights reserved.

puts "PGS Test"
puts "=========="
10.times do |i|
  puts "Test number #{i}"
  result = ‘/usr/bin/time -lp ./sets-pgs.exe testcase1.input >
    testcase-pgs.output‘
end

puts "COLA Test"
puts "=========="
10.times do |i|
  puts "Test number #{i}"
  result = ‘/usr/bin/time -lp ./sets-cola.exe testcase1.input >
    testcase-cola.output‘
end

puts "Bison Test"
puts "=========="
10.times do |i|
  puts "Test number #{i}"
  result = ‘/usr/bin/time -lp ./sets-bison.exe testcase1.input >
    testcase-bison.output‘
end
```

5.4.2 Test Input

The grammar generates from a simple definition of sets of names a C-like set specification. For generating a large test case I wrote a ruby script, which generated an input file for the sets translator.

```
#!/usr/bin/env ruby
#
#  Created by Ulf Schwekendiek on 2007-07-20.
#  Copyright (c) 2007. All rights reserved.

def genTestCase1(file, x,y)

  def newrandom( len )
    chars = ("a".."z").to_a + ("A".."Z").to_a
    random = ""
    1.upto(len) { |i| random << chars[rand(chars.size-1)] }
    return random
  end

  theFile = File.new(file, "w")
  line = ""

  #generate x sets
  x.times do |i|
    #set name
    line = line + newrandom(rand(10)+5)+"{"
    y.times do
      line = line + newrandom(rand(10)+5) + " "
    end
    line = line + "}"
    theFile.puts(line)
    line = ""
    puts i.to_s +  "/" + x.to_s
  end
  theFile.close
end

genTestCase1("testcase1.input",500,1000);
genTestCase1("testcase2.input",1000,2000);
```

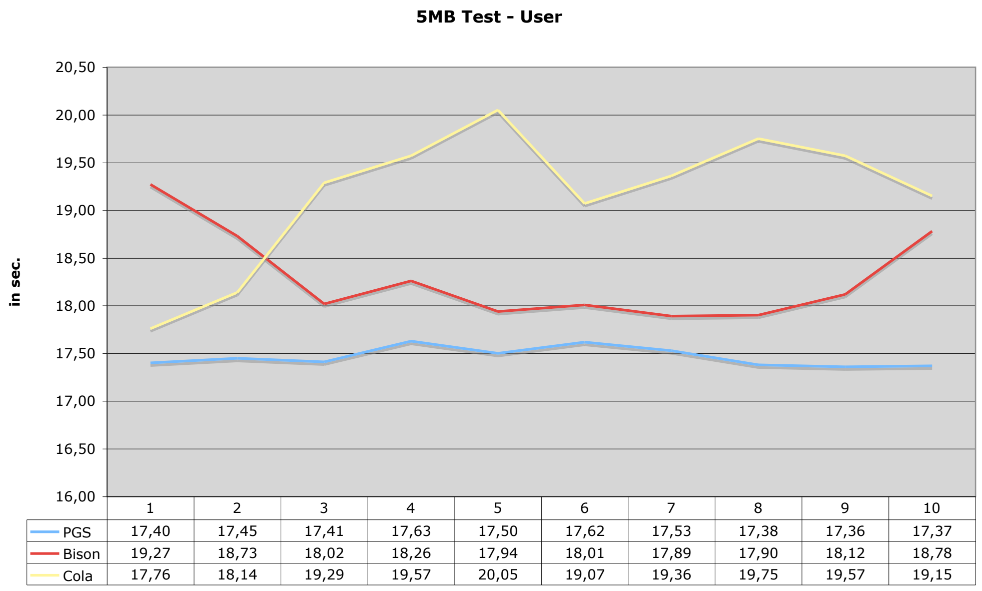
This programm generates a 5 MB and a 20 MB input file.

5.4.3 Test Results

The first test needed on an iMac G5 with 1.8GHz PowerPC processor on the 5MB file in difference 180 seconds for each turn, while the same test needed on a

MacBook with a 2.3 GHz Core2Duo processor in difference 18 seconds. The 20 MB test took in difference 465 seconds on the MacBook. The generated test graphs in this section are from the MacBook test.

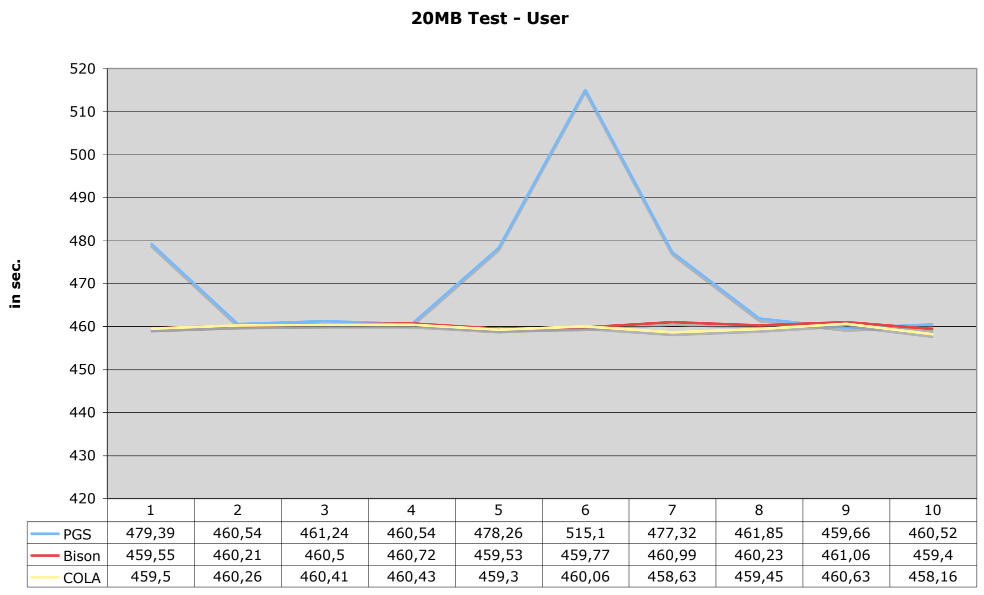
Figure 5.1 shows, that the Bison execution time is between the execution time from Pgs and Cola. Therefore there is no great execution time difference diagnosed. However as bigger the input will get (see figure 5.2) the time difference between the parsers shrinks. The test lets me reason, that the Bison parser is not slower than the other generated parsers.



	1	2	3	4	5	6	7	8	9	10
PGS	17,40	17,45	17,41	17,63	17,50	17,62	17,53	17,38	17,36	17,37
Bison	19,27	18,73	18,02	18,26	17,94	18,01	17,89	17,90	18,12	18,78
Cola	17,76	18,14	19,29	19,57	20,05	19,07	19,36	19,75	19,57	19,15

Figure 5.1: 5 MB test case - User time

The complete diagrams and tables for real, user and system time for both test cases are located in the appendix.



	1	2	3	4	5	6	7	8	9	10
PGS	479,39	460,54	461,24	460,54	478,26	515,1	477,32	461,85	459,66	460,52
Bison	459,55	460,21	460,5	460,72	459,53	459,77	460,99	460,23	461,06	459,4
COLA	459,5	460,26	460,41	460,43	459,3	460,06	458,63	459,45	460,63	458,16

Figure 5.2: 20 MB test case - User time

6 Conclusion

To recapitulate, it was possible to add the Bison parser generator as a tool to the Eli package and expand therefore the possible types of grammars, that can be parsed with that new tool. The parser generator was properly implemented and tested, so it can be said, that it is an improvement for the Eli system.

Also the speed of the generated parser is as fast as the speed of the other parsers. However because of the missing error recovery it is only appropriate to use it for special points at this time being.

6.1 Future Work

In this section future work on this subject is explained.

6.1.1 Error Recovery

At this point, if the Bison generated parser has a parsing error, the error is outputted and the parsing process stops. For a good parser this is not acceptable and better error recovery methods could be added to the grammar. This could be done in a first simple stage automatically by the tool `maptool`. To any productions like

```
A : B '+' C
  | B '-' C
  ;

B : id
  |   num
  ;

C : id
  ;
```

a new alternative production with `error` as the only right hand side token has to be added. Also a semantic action for that method has to be added in order to do error recovery. If another terminal symbol after the `error` symbol is added, the Bison generated parser automatically skips all tokens until this symbol is parsed. Also important for both ways is that in the semantic action the macro `yyerrok;` is included for an immediate parse resume.

For the above example this would be a possible solution:

```
A : B '+' C
  | B '-' C
  | error ';' { yyerrok; }
  ;

B : id
  |   num
  | error { yyerrok; }
  ;

C : id
  | error ';' { yyerrok; }
  ;
```

6.1.2 Version Checking in Installation

Another point that is not implemented in my Eli version is a Bison version checking in the installation process. A new autoconf function has to be written, which extracts the version number from `bison --version` and needs to make sure that the Bison version is able to generate GLR parsers.

6.1.3 Ambiguous Grammars

In a Bison generated GLR parser, even if not all parsing branches merge (see figure 6.1), the parser does not exit with a parser error.

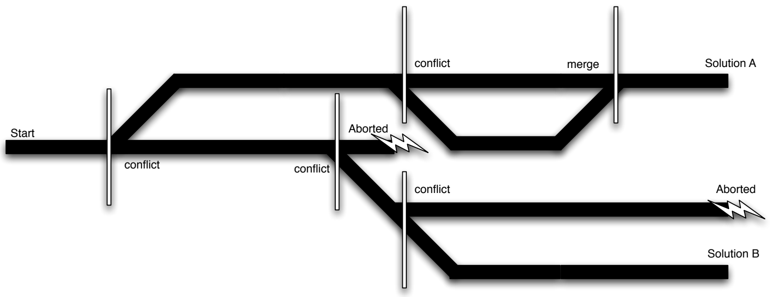


Figure 6.1: Example parse trace of a GLR parser, where two different parse paths have a valid solution. This can happen in an ambiguous grammar.

In this case a Bison generated parser would execute both semantic actions, which are in the Eli case commands for tree construction. This causes a serious problem, because if both actions are executed, an error occures in the parse tree construction and the whole tree can not be traversed. However Bison does not inform the user, that the given grammar is ambiguous, but it supports a special "merge" function, which can be defined in such a case. Another solution Bison supports is to give

priorities to each parsing tree.
Both solutions are not usable by Eli directly, because the Eli translator designer has no direct access to the Bison specification file, where these solutions can be specified.

Here are now two possible solutions for this problem:

- **Stepping away from pgram grammar and writing instead Bison specification files directly.**
  Here an interface to the scanner and to the tree translation has to be built. Therefore the tool `maptool`, which does that processes right now, can not be used anymore and has to be written completely new.

- **Adding some special Bison commands in the pgram grammar.**
  This solution would be more maintainable. Some Bison special commands for merging or priorities in GLR parser could be added to pgram. Here it is important to observe whether Bison is chosen as the parser generator or Pgs or Cola. The changes have to be added to the `maptool` program.

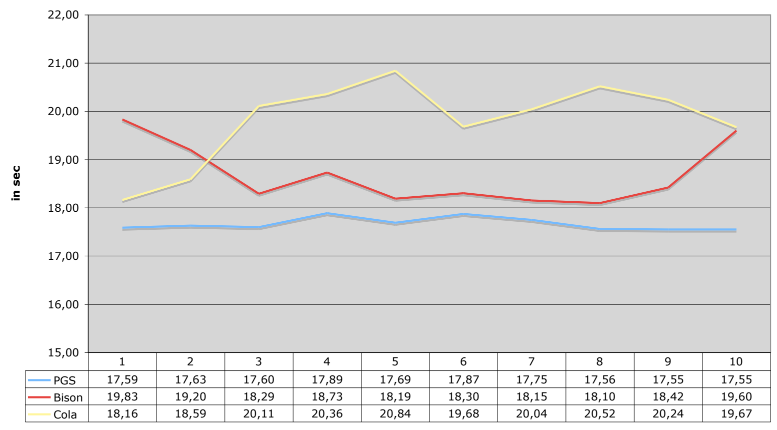
6.1.4 Bison Special Parsing Declarations

Another possible idea would be to add a special property file for the Bison generated parser in which declarations such as operator precedence for resolving ambiguities can be defined. This specification file should be merged with the generated Bison specification. It is not clear that the methods for operator precedence are working properly with the generated semantic actions for the tree generation.

Appendix

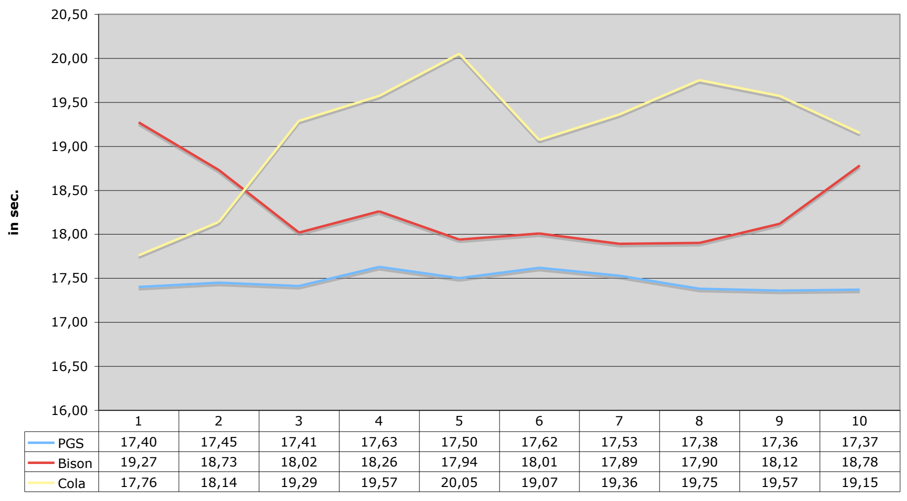
5 MB Test

5MB Test - Real



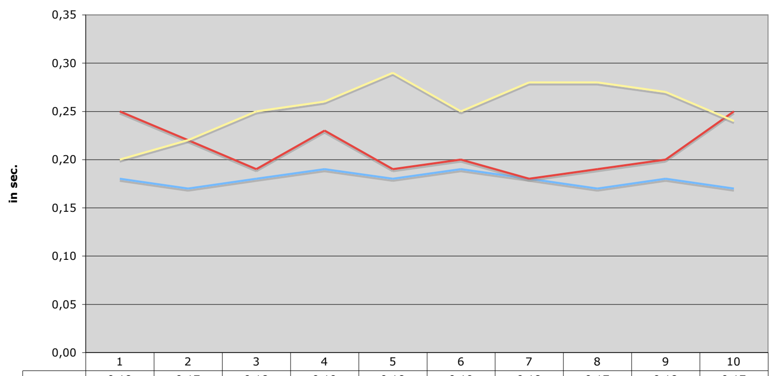
	1	2	3	4	5	6	7	8	9	10
PGS	17,59	17,63	17,60	17,89	17,69	17,87	17,75	17,56	17,55	17,55
Bison	19,83	19,20	18,29	18,73	18,19	18,30	18,15	18,10	18,42	19,60
Cola	18,16	18,59	20,11	20,36	20,84	19,68	20,04	20,52	20,24	19,67

5MB Test - User



	1	2	3	4	5	6	7	8	9	10
PGS	17,40	17,45	17,41	17,63	17,50	17,62	17,53	17,38	17,36	17,37
Bison	19,27	18,73	18,02	18,26	17,94	18,01	17,89	17,90	18,12	18,78
Cola	17,76	18,14	19,29	19,57	20,05	19,07	19,36	19,75	19,57	19,15

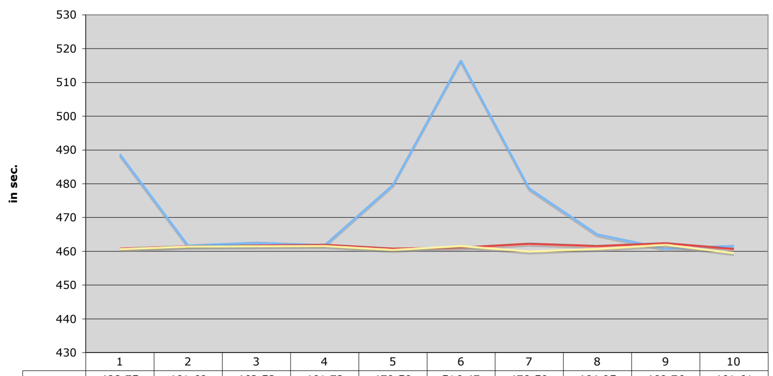
5MB Test - System Time



	1	2	3	4	5	6	7	8	9	10
PGS	0,18	0,17	0,18	0,19	0,18	0,19	0,18	0,17	0,18	0,17
Bison	0,25	0,22	0,19	0,23	0,19	0,20	0,18	0,19	0,20	0,25
Cola	0,20	0,22	0,25	0,26	0,29	0,25	0,28	0,28	0,27	0,24

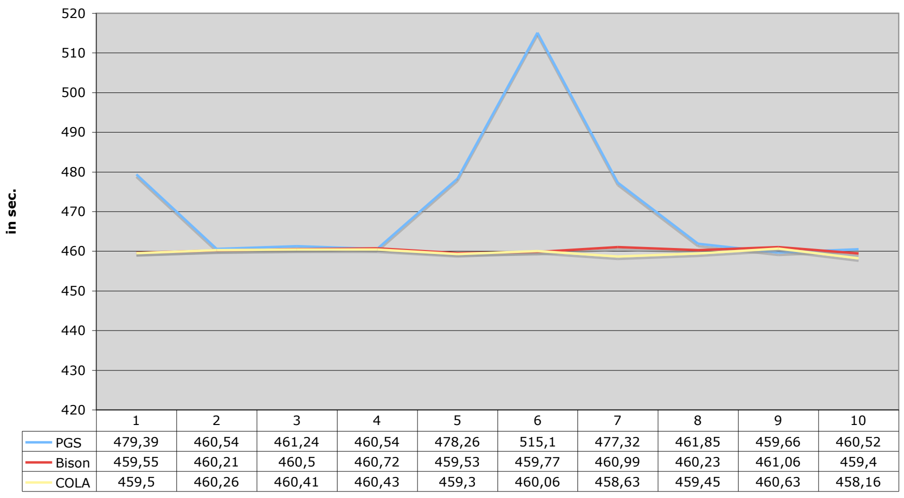
20 MB Test

20MB Test - Real



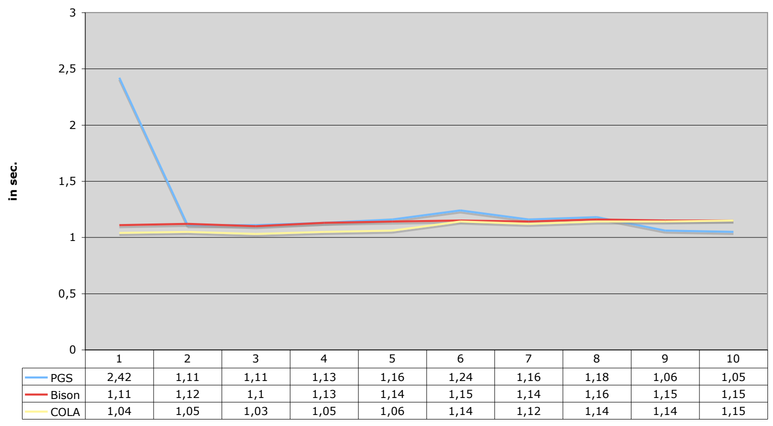
	1	2	3	4	5	6	7	8	9	10
PGS	488,75	461,69	462,52	461,72	479,59	516,47	478,59	464,95	460,76	461,61
Bison	460,75	461,42	461,69	461,94	460,78	461,03	462,22	461,52	462,33	460,66
COLA	460,59	461,34	461,48	461,52	460,4	461,5	459,87	460,7	461,87	459,42

20MB Test - User



	1	2	3	4	5	6	7	8	9	10
PGS	479,39	460,54	461,24	460,54	478,26	515,1	477,32	461,85	459,66	460,52
Bison	459,55	460,21	460,5	460,72	459,53	459,77	460,99	460,23	461,06	459,4
COLA	459,5	460,26	460,41	460,43	459,3	460,06	458,63	459,45	460,63	458,16

20 mb test - System



	1	2	3	4	5	6	7	8	9	10
PGS	2,42	1,11	1,11	1,13	1,16	1,24	1,16	1,18	1,06	1,05
Bison	1,11	1,12	1,1	1,13	1,14	1,15	1,14	1,16	1,15	1,15
COLA	1,04	1,05	1,03	1,05	1,06	1,14	1,12	1,14	1,14	1,15

List of Figures

Bibliography

[1] http://rw4.cs.uni-sb.de/~sander/html/gsvcg1.html last checked at 23.06.2007.

[2] http://www.cygwin.com/ last checked at 20.07.2007.

[3] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design (Addison-Wesley series in computer science and information processing)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1977.

[4] Geoffrey M. Clemm. *The Odin Reference Manual*, version 1.17 edition.

[5] Compiler Tools Group UoC, University of Colorado - Departement of Electonical and Computer Engineering. *Guide for New Eli Users - Revision 3.10*, 2002.

[6] Charles Donnelly and Richard Stallman. *Bison 2.3 - The Yacc-compatible Parser Generator*. Free Software Foundation, 51 Franklin Street, Fifth Floor Boston, MA 02110-1301 USA, May 2006. http://www.gnu.org/software/bison/manual/index.html.

[7] Uwe Kastens. Programming languages and compilers, course slides. http://ag-kastens.uni-paderborn.de/lehre/material/plac/.

[8] Kenneth C Louden. *Compiler Construction - Principles and Practice*. PWS Publishing Company, a division of International Thomson Publishing Inc., 1997.

[9] Elizabeth Scott, Adrian Johnstone, and Sadaf Hussain. Tomita-style generalised lr parsers. *Royal Holloway University of London*, 2000.

[10] M. Tomita and S.-K. Ng. The generalized lr parsing algorithm. In M. Tomita, editor, *Generalized LR Parsing*, pages 1–16. Kluwer, Boston, 1991.