Comparing Generators for Language-based Tools

Daniela da Cruz, Maria João Varanda, Mario Berón, Rúben Fonseca, and Pedro Rangel Henriques

Departamento de Informática
Universidade do Minho, CCTC, Braga, Portugal,
{danieladacruz|rubenfonseca}@di.uminho.pt

Abstract. The first step in any language development project is the Compiler Generator choice. Nowadays there are many offers, based on translation grammars, attribute grammars or other language specification formalisms. To make up a decision, more factors than just the tool user-friendliness and the processor's quality should be taken into account. To aid the language developer, we analyze in this paper three Compiler Generators. The traditional and well known YACC, and two more recent ones, LISA and AnTLR-3. The first produces a Syntax-Directed Translator, while the others generate a Semantic-Directed Translator based on attribute evaluation. Moreover both the AG-based generators also produce other Language-based Tools that are mentioned and compared.

1 Introduction

To start the development of a new language (a general-purpose programming language (PL), or a domain-specific language (DSL)), one of the first steps is the selection of the tool—the Compiler Generator (CG)—that will be used to build the language processor (language analyzer and transformer).

In fact the CG choice imposes the formalism to be used for language specification (a translation grammars (TG) or an attribute grammar (AG)), the kind of parser (top-down (TD) or bottom-up (BU)), the quality/efficiency of the generated processor, and also influences the overall development time.

At moment, CGs (that will be discussed in section 2) are more than simple non-interactive compiler compilers; as will be argued in subsection 2.1, most of them are truly language development laboratories (environments), providing editors for the meta-language, and analyzers to make easier grammar debugging and improvement. Moreover, some of them also are able to generate Language-based Tools (see also subsection 2.1), to aid the language end-user.
The availability of that paraphernalia should, in our opinion, be taken into account during the decision process.

Faced with this scenario, we decided to study different CGs and identify the *pros and contras*. For that purpose we chose a DSL, called hereafter Lavanda and introduced in section 3, that will be used as our case-study. We developed a full processor for Lavanda using each one of the tools under study, as extensively reported in [dCH06].

For this paper we elected three Compiler Generators. The traditional and well known YACC (see section 4)—a Syntax-Directed Translator that accepts as input a Translating LALR Grammar with just 1 synthesized attribute—and two more recent ones: LISA (see section 5) and AnTLR-3 (see section 6). Both accept attribute grammars, LL or LALR the first and LL(*) the second, and generate a Semantic-Directed Translator. Moreover, both provide grammar depuration utilities and generate complementary, end-user oriented, language-based tools that are mentioned and illustrated.

However, it is not our intention to conclude which one is the best; definitively, we will not fight for one of those systems. We just want to give directions that could help the CGs' user in his decision-making process; we will show a criteria that he can use to choose quickly a tool that better fulfill the requirements of his concrete problem, and that better fits into his programming skins, style and background.

In the conclusion (section 7), a comparison table is included to resume and systematize the lessons taken from our exploration. The criteria used to build that table is based on a set of parameters that were divided into 3 subsets: the first one is concerned with *language specification*; the second one assesses characteristics of the *generated compiler*; and the third deals with characteristics of the *compilers generator* itself.

2 Language Processors and Automatic Generation

The *basic tool to process a language* (generally speaking, a *compiler*) is built up from a scanner (for lexical analysis), a parser (for syntactic analysis), a semantic analyzer (to evaluate the meaning and check the semantic constraints), and a code generator (to produce the output). This compiler just analyzes the source language sentences and transforms (translates) them into the desired output (whatever it is).

The development of the first compilers in the late fifties without adequate tools was a very complicated and time consuming task. Later on, formal methods, such as operational semantics, denotational semantics, action semantics, algebraic semantics, and attribute grammars, were developed. They made the implementation of programming languages easier and finally contributed to the automatic generation of language processor. The programs responsible for that automatic production we will be called, from now on, *Compiler Generators* (CG).

The standard definitions about languages and context-free grammars, that make automatic implementation of programming languages possible, can be found in classical textbooks, such as [HU79,WG84,ASU86]. To specify the semantics of programming languages, context-free grammars need to be extended. Among the several possibilities, we adopted the attribute grammar approach. Attribute grammars [Knu68] are a generalization of context-free grammars in which each symbol has an associated set of attributes that carry semantic information; associated with each production a set of semantic rules specify the attribute value computation. Attribute grammars have proved to be very useful in specifying the semantics of programming languages, in automatic construction of compilers/interpreters, and in specifying and generating interactive programming environments [Paa95].

2.1 Language-based Tools

The formal definition of languages allows to construct the above referred compiler components—scanners, parsers, semantic evaluators, and code generators—in a systematic way. As already told, it also supports their automatic construction. Nowadays, researchers recognize the possibility of use formal language specifications to generate other language-based tools [HVM$^+$05].

There are two different kinds of language-based tools that can be generated: *tools aimed at the language end-user*, to help him editing and analyzing source texts written in that language; and *tools to aid the compiler developer* editing and optimizing the grammar for the language.
For the language end-user, some examples of tools are: *pretty printers, structured editors, syntax-directed editors, data flow analyzers, partial evaluators, debuggers, profilers, test case generators, visualizers, animators, documentation generators*, etc.
For the language developer, some examples of tools are: *language specification editors* (this is, grammar editing environments); and *language processing inspectors.* In the second group we include tools like: visualizers for regular expression diagrams, syntax diagram builders, attribute dependency graph builders, lexical automaton visualizers, syntax tree visualizers, semantic (or attribute) tree visualizers, parser debuggers, attribute evaluation animators, etc.

In some cases, the language specification is enough to generate these tools. In other cases, the language specification must be extended, or appropriate information extracted, in order to be able to automatically generate a language-based tool. The implementation of these language-based tools ([HVML02]) is divided into a *generic fixed part* that uses pre-defined algorithms to traverse the internal data structure that is generated by the *variable part* (source language dependent). This kind of approach avoids useless time consuming and it decreases the number of errors which can turn maintenance less expensive.
We can also take profit of this approach when we want to implement DSLs because it is not common to develop language-based tools like debuggers or pretty printers for these kind of languages; usually just a compiler(or interpreter) is constructed.

3 Lavanda, our case-study

In this section we will introduce the case-study: Lavanda language implementation.

3.1 The problem

Lavanda is a Domain Specific Language (*DSL*) whose main goal is to describe the laundry bags (Sacos) that the collection point (IdPR) of a big launderette company daily sends to the central building to wash—we call it the ON *ordering note*.
Each bag (Saco) is identified by an id number, num, and a client name, IdCli; its content is composed by one or more items (Lotes). Each item (Lote) is a subset of laundry of the same type. The type is characterized by: a laundry class, Classe, (corpo, *body cloth*, or casa, *household linen*); a kind of tinged, Tinto, (br, *white*, or cor, *colored*); and a raw-material, Fio (alg, *cotton*, la ,*wool*, or fib, *fiber*). For each one item, we register the number of pieces collected.
The (abstract) Context Free Grammar below defines the syntax of the desired language Lavanda. The root of the grammar is Lavanda. Terminal symbols are written in lowercase (pseudo-terminals), uppercase (reserved-words), or between apostrophes; the remaining symbols are Non-Terminals. Notice that the concrete grammar is similar, but does not matter for our purpose (semantic specification).

```
p1:          Lavanda →  Cabec  Sacos
p2:          Cabec   →  date  IdPR
p3:          Sacos   →   Saco  '.'
                     |  Sacos  Saco  '.'
p5:          Saco    →   num  IdCli  Lotes
p6:          Lotes   →   Lote
                     |  Lotes Lote
p8:          Lote    →   Tipo  Qt
p9:          Tipo    →   Classe  Tinto  Fio
p10:         IdPR    →   id
p11:         IdCli   →   id
p12:         Qt      →   num
p13,14:      Classe  →   corpo |  casa
p15,16:      Tinto   →   br    |  cor
p17,18,19:   Fio     →   alg   |  la    |  fib
```

The **problem** consists in *processing an ON*—analyze the laundry bags list, check that there are no two bags with the same id number or the same client name, and compute some numbers. Namely, the demand requires the computation of:

1. the amount of bags received during the day in the collecting point;
2. the total of items per client (the number of items in each bag);
3. the total of pieces per type (for each one of the 12 item types, ranging from 'corpo/br/alg' until 'casa/cor/fib');
4. the cost per client (given the price of each laundry type).

For the purpose of this paper, we will forget the semantic checking and we just will consider the first 3 items above.

3.2 Attribute Grammar

To solve the problem using an attribute grammar, we take the CFG above as the structural or syntactic basis, and then it is necessary to proceed in 2 steps: (a) choose the attributes (their name, type and class[1]) to associate with each grammar symbol in order to handle all the information needed to compute the required results; (b) write down the attribute evaluation and translation rules, and the contextual conditions associated with each production (grammar rule).

We advocate an incremental approach to AG development. It means that those 2 steps should be applied successively to solve each problem requirement. In our case we execute steps a) and b): four times, to specify the evaluation/translation rules necessary to compute the 4 values demanded; another one, to specify the semantic constraints (i.e. include the contextual conditions). Below, 3 different colors (blue, green and red) are used to distinguish the 3 phases related with the 3 requirements that we will care about.
After that we merge the partial AGs, obtained so far, to produce the solution for the problem.
Table 1 assembles all the attributes chosen, gathering the outcome of step a) for the three phases.

Item	Att-Name	Att-Type	Att-Class	Symbols
1	nSacos	int	syn	Lavanda, Sacos
2	nLotes	int	syn	Saco, Lotes
3	inEnv	HashTable	inh	Saco, Lotes, Lote
	outEnv	HashTable	syn	Lavanda, Sacos, Saco, Lotes, Lote
	name	String	syn	Tipo, Classe, Tinto, Fio

Table 1. Attributes used in Lavanda AG

The attribute evaluation and translation rules, necessary to solve sub-problem 1 to 3, are shown below after merging them (notice the colored schema referred above).

```
p1:        Lavanda.nSacos  =  Sacos.nSacos;
           println( Lavanda.nSacos );
           println( Sacos.outEnv )

p3:        Sacos.nSacos   =    1;
           Saco.inEnv  =  Sacos.inEnv;
           Sacos.outEnv  =  Saco.outEnv

p4:        Sacos0.nSacos  =    Sacos1.nSacos + 1
           Saco.inEnv  =  Sacos1.outEnv;
```

[1] An attribute should be either *inherited*, or *synthesized*.

```
           Sacos1.inEnv  =  Sacos0.inEnv;
           Sacos0.outEnv  =  Saco.outEnv

p5:        Saco.nLotes   =   Lotes.nLotes;
           println( Saco.nLotes );
           Lotes.inEnv = Saco.inEnv;
           Saco.outEnv = Lotes.outEnv

p6:        Lotes.nLotes  =   1
           Lote.inEnv = Lotes.inEnv;
           Lotes.outEnv = Lote.outEnv

p7:        Lotes0.nLotes  =   Lotes1.nLotes + 1;
           Lote.inEnv = Lotes1.outEnv;
           Lotes1.inEnv = Lotes0.inEnv;
           Lotes0.outEnv = Lote.outEnv

p8:        Lote.outEnv = updateTable(Lote.inEnv,
Tipo.name, num)

p9:        Tipo.name = Classe.name ++ Tinto.name ++ Fio.name

p13:       Classe.name = "corpo"
p14:       Classe.name = "casa"
p15,...,19:........
```

Due to space limitations, we elected just three non-terminals—`Lotes`, `Sacos` and `Lote`—and the respective five productions—`p3/p4`, `p5/p6` and `p8`—to be discussed in rest of the paper.

3.3 Source-Text: a small example

We show below the source-text that will be submitted for processing to the tool under study in each of the next three sections. the following one:

```
10-11-2007 Carrefour
           1 ClientA (corpo-cor-la 1 , casa-cor-alg 2)
           2 Clientb (corpo-cor-fib 10 )
```

This input will be used in all the examples along the paper.

4 YACC

YACC is a traditional tool aimed at building syntactic analyzers. YACC is considered the more successful tool in the *compiler generation* context and was used to build many important system, mainly compilers. Between the systems built using

YACC we can found the compilers for languages C, Pascal, APL, RATFOR, etc. YACC also was used for less conventional languages such as photo-typesetter language, several desk calculator languages and document retrieval system. For this reason YACC is an important tool to be taken as reference point to analyze and compare another approaches used for recent tools. Before illustrating its usage with our case-study, Lavanda, in subsection 4.2, we present in the next subsection a very short description of this tool. Then we complement that brief introduction with some comments on *grammar depuration aids* provided by YACC when the appropriate switches are inserted in the invocation line (subsection 4.3); no Language-based Tools are generated to help the language end-user.

4.1 A brief introduction to YACC

YACC is a computer program aimed at generating syntax analyzers (parsers). Its name is an acronym for Yet Another Compiler-Compiler. The software generated by YACC is a parser—the part of the compiler that recovers the structure of the input text or reports an syntactic error, according to a given grammar. YACC generates the code for the parser in C programming language. YACC was developed by Stephen C. Johnson [Joh75] at AT&T for the Unix operating system.

YACC requires a lexical analyzer. For this reason this tools is often used with a lexical analyzer generator, commonly Lex [LS75][2]; this statement can be confirmed anywhere, for instance The Lex & Yacc Page at `http://dinosaur.compilertools.net`.
In order to use YACC the user prepares a language specification. This specification (a Translation Grammar) contains rules that describe the input structure, and code to use when this rules are recognized. With this information YACC generates a function, called `yyparse()`. The parser invokes the lexical analyzer to get the next token from the input stream. When the accepted tokens can be reduce to recognize a production, the code defined by the user for this rule is executed.
Sometimes YACC fail to produce the parser; normally this situation happens because there are contradictory specifications or the given grammar requires a more powerful recognition mechanism than the LALR provided by YACC.

The user can associate semantic actions with each one grammar production. A semantic action is a set of C statements that can make input-output, call subprograms, etc. The semantics actions are specified by C statements closed between { and }. To facilitate the communication between the semantic action and the parser YACC use the symbol $. To return a value the semantic actions set the pseudo-variable $$ with the wished value. On the other hand to obtain the values produced by previous actions the actions can be use the pseudo-variables $1, $2, ...,$n,.. YACC uses the shift-reduce algorithms to parse the input stream.

YACC has different mechanisms to handle ambiguity problems, precedence, error handling, etc. All this mechanism are well known and a good reference can be found in [LMB92].

[2] Nowadays also available at `http://epaperpress.com/lexandyacc/download/lex.pdf`.

4.2 Lavanda implementation

In this subsection we present a simple and short example of a YACC implementation for two Lavanda Language productions. The reader can observe that the first productions are left recursive. It is important because avoid the problems mentioned in the previous subsection. Furthermore is possible to analyze as the user can use the pseudo-variables and write the semantics actions in one concrete case. It is important emphasize how the synthesized attributes are implemented in YACC using the pseudo-variables.

```
Sacos:
       Saco '.'        {$$=1;}
     | Saco Saco '.'   {$$=$1 + 1;}

Lotes:
       Lote            {$$=1;   }
       Lote Lotes      {$$=$2+1;}
     ;

Lote:
       Tipo NUMBER      { updateTableNLotes(inEnv,$1,$2); }
     ;
```

The attribute used in symbols `Saco` and `Sacos` is an synthesized attribute (the `nSacos` and `nLotes` referred in section 3.2) and the attribute used in symbol `Lote` is the attribute where the table is updated.

4.3 Generated Tools

The YACC output consists of one C program that implements one syntax analyzer LALR. YACC can be invoked with different parameters that allows to generate some facilities, described below, oriented to the language implementor:

1. When YACC is invoked with the parameter -r, it produces separated files for the parsing tables and code. The code is named `y.code.c` and the tables file is named `y.tab.c`.
2. when YACC is invoked with the parameter -t, it incorporates debugging statements in the generated code (*code instrumentation*).
3. When YACC is invoke with the parameter -v, it produces some information in human-readable format, more precisely the LALR automaton.

Concerning the generation of tools to help the language end-user, YACC has nothing to offer; we must accept that this evaluation criterion reveals the biggest YACC weakness.

5 LISA

LISA (Language Implementation System based on Attribute grammars) is a compiler-compiler, or a system that generates automatically a compiler/interpreter from formal AG-based language specifications.

5.1 A brief introduction to LISA

Some years ago, Marjan Mernik, and his team at Maribor University conceived, designed and implemented a new compiler generator supporting object-oriented attribute grammars [MKŽ95,MLAŽ98]. His system, called LISA, is a generic interactive environment [MLAŽ02] for programming language development. From the formal language specifications of a particular programming language, LISA produces a set of related tools described in subsection 5.3. LISA and the generated environment are written in JAVA which enables high portability to different platforms [MNA^{+}01].

LISA specification language provide construction for: regular expression definitions (lexical part); attributes definitions and rules — which are generalized syntax rules (described with the context free grammars using a variant of the BNF notation) that encapsulate semantic rules and methods (written in JAVA).

The set of syntax rules form a generalization of context-free grammars, designed AG, in which each symbol has an associated set of attributes with semantic information, and each production have associated a set of semantic rules with attribute computation (this semantic rules are JAVA assignment statements).

LISA syntax rules are written in a block { } which start with keyword compute. Each rule has unique name and starts with reserved word rule.

```
rule RuleName {
NAME ::= DEFINITION compute {
              <attribute computation>
           };
}
```

Being a truly AG-based compiler generator and generating a set of related tools, LISA provide us a system capable to develop a language using synthesized and inherited attributes, and also trace attribute computation. LISA compiler is also capable of generating $LR(0)$,$LR(1)$, $LL(1)$, $LALR(1)$ and $SLR(1)$ parsers.

5.2 Lavanda implementation

In this subsection, we present a part of the Lavanda implementation. In order to divide the implementation of each exercise of the 2 rules produced in subsection 3.2, we will use a feature of LISA: multiple attribute grammar inheritance. *Multiple attribute grammars inheritance is a structural organization of attribute grammars where the attribute grammar inherits the specification from ancestor attribute grammars, may add new specifications or may override some specifications from ancestor specifications.*[MZLA99]

So each exercise, will be traduced by an attribute grammar, where the first one is the base of the two others. To implement this, we just give to LISA the import of the precede grammar and point, in each rule, that is an extension of the previous one.

– Computation of number of bags and items:

```
rule Sacos {
        SACOS ::= SACO compute
            {   SACOS.nSacos = 1;       }
            | SACOS SACO
            {   SACOS[0].nSacos = SACOS[1].nSacos + 1; };
}

rule Lotes {
        LOTES ::= LOTE compute
            { LOTES.nLotes = 1;   }
            | LOTES LOTE
            { LOTES[0].nLotes = LOTES[1].nLotes + 1;  };
}
```

The attributes used are:

nSacos::int : in case of the production derive in one bag we assign `nSacos` to 1; otherwise we use the attribute synthesized in occurrence of `SACOS` at right side (`SACOS[1]`) and add 1 bag;

nLotes::int : same procedure of `nSacos`.

– Compute and print the number of pieces that belongs at each one of 12 items type:

```
rule extends Sacos {
        SACOS ::= SACO compute
            {
                SACO.inTable   = SACOS.inTable;
                SACOS.outTable = SACO.outTable;
                }
            | SACOS SACO compute
            {
                SACO.inTable      = SACOS[1].outTable;
                SACOS[1].inTable  = SACOS[0].inTable;
                SACOS[0].outTable = SACO.outTable;
            };
}

rule extends Saco {
        SACO ::= #Number #Identifier \( LOTES \) compute
        {
                LOTES.inTable  = SACO.inTable;
                SACO.outTable  =  LOTES.outTable;
        };
}


rule extends Lotes {
        LOTES ::= LOTE compute
             {
                 LOTE.inTable    =  LOTES.inTable;
                 LOTES.outTable  =  LOTE.outTable;
             }
             | LOTES LOTE compute
             {
                LOTE.inTable      = LOTES[1].outTable;
                LOTES[1].inTable  = LOTES[0].inTable;
                LOTES[0].outTable = LOTE.outTable;
             };
}
```

To compute the number of pieces of each type we will need 3 attributes:

inEnv::HashTable : inherited attribute to keep each type of bags already read (initially, the elements of this table is initialized with 0 — in root production with semantic action: `SACOS.inEnv = initNLotes();`). When a new bag is read, this attribute is changed and passed in `outEnv`;

outEnv::HashTable : synthesized attribute used to compute the number of pieces recognized;

name::String : synthesized attribute to compute the new type of piece read (`corpo/br/alg`, ...) and associated at `Classe`, `Tinto` and `Fio` productions.

In this productions, the attribute `(in|out)Env` is used to reflect the changes in table of types of pieces. This attribute is passed from left to right. In case of just one bag or item, the synthesized table is that one that result from changing in `Lote` production. The "bridge" between `Sacos` and `Lotes` productions is `Lote` production with semantic action:

```
LOTE.outEnv = updateEnv(LOTE.inEnv,TIPO.name,#Number.value());
```

In case of 2 or more bags or items, the synthesized attribute is the result obtained at `Saco`, after this production had received the result of synthesized attribute at `Sacos[1]`.

5.3 Generated Tools

In this subsection we will present the generated tools by the compiler-compiler LISA. The source program written obeying the rules of Lavanda language and used to generate the figures showed at this part is the referred in section 3.

LISA display us a set of related tools aiming at understand behaviour of the generated compiler from a source language specified in LISA. The families of such tools are [MLAZ00,Rep82,HVML02]:

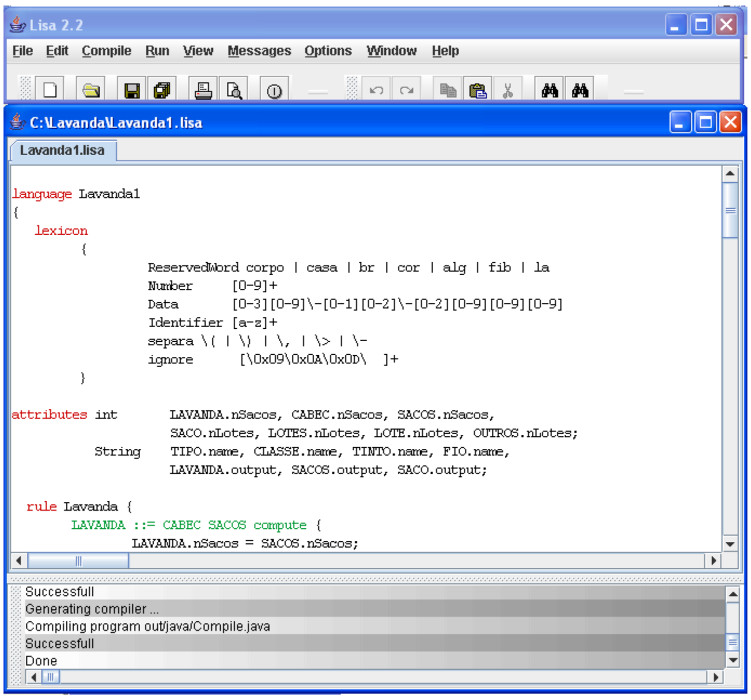
Editors : to help the users in the creation and maintenance of the specified language. The main editor is a language knowledgeable editor, that represents a compromise between text editors and syntax-directed editors, from formal language specifications (figure 1);

Inspectors for language processors : regular expressions in LISA have a visual representation like directed graphs using *Finite Site Automata* (figure 2 the finite automata of Lavanda is presented); a graphical browser for the *Syntax Tree* built after the generated compiler parse a given source program (figure 3 illustrates this tool); and a *Dependency Graph* to analyze the order of the attribute evaluation (figure 4).

Semantic Evaluator Animation : this evaluator animate the visits to the nodes of the semantic tree and the evaluation of attributes of these nodes. This evaluator can also be helpful in debugging process (figure 5).

BNF viewer : this generated tools able us to see our grammar in Backus-Naur Form (figure 6).

Follow and First symbols : LISA is also able to compute the first and follow symbols of one given source language specification (figure 7).



```
language Lavanda1
{
   lexicon
         {
                  ReservedWord corpo | casa | br | cor | alg | fib | la
                  Number     [0-9]+
                  Data       [0-3][0-9]\-[0-1][0-2]\-[0-2][0-9][0-9][0-9]
                  Identifier [a-z]+
                  separa \( | \) | \, | \> | \-
                  ignore      [\0x09\0x0A\0x0D\  ]+
         }

attributes int       LAVANDA.nSacos, CABEC.nSacos, SACOS.nSacos,
                     SACO.nLotes, LOTES.nLotes, LOTE.nLotes, OUTROS.nLotes;
           String    TIPO.name, CLASSE.name, TINTO.name, FIO.name,
                     LAVANDA.output, SACOS.output, SACO.output;

  rule Lavanda {
        LAVANDA ::= CABEC SACOS compute {
                LAVANDA.nSacos = SACOS.nSacos;
```

```
Successfull
Generating compiler ...
Compiling program out/java/Compile.java
Successfull
Done
```

Fig. 1. LISA editor environment

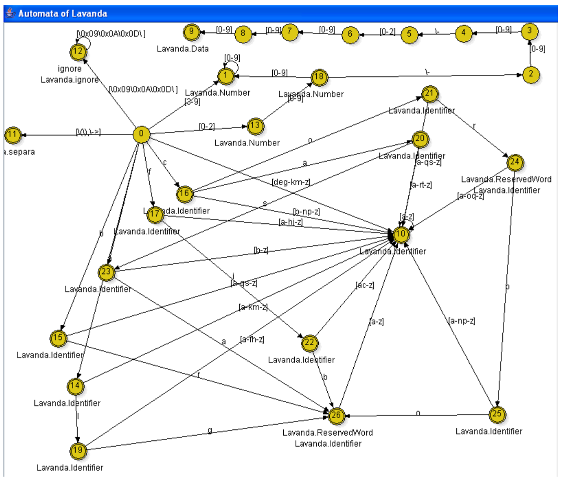


Fig. 2. FSA to set of regular expressions defined in Lavanda

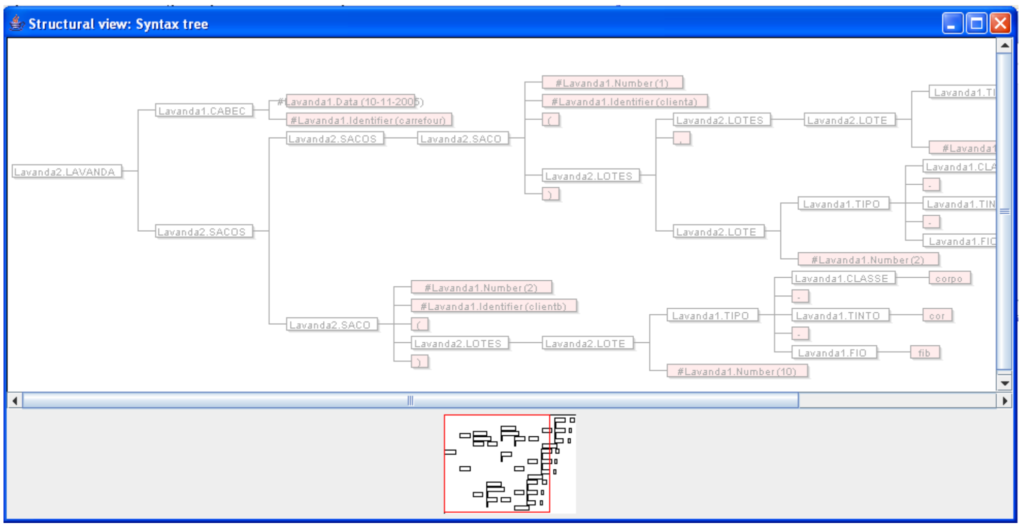
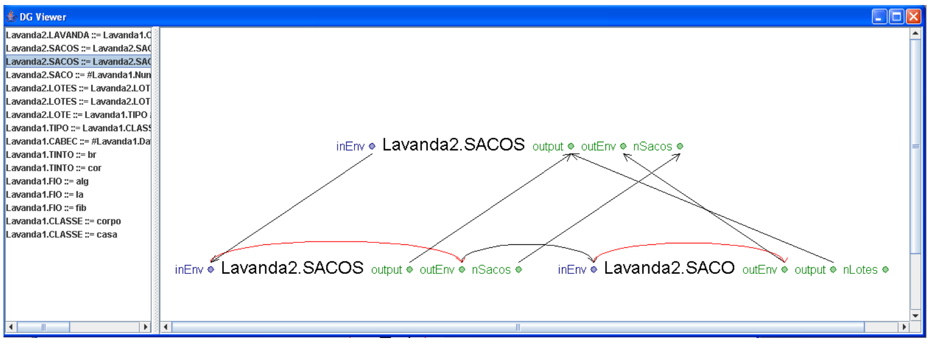


Fig. 3. Syntax tree of Lavanda



```
Lavanda2.LAVANDA ::= Lavanda1.C
Lavanda2.SACOS ::= Lavanda2.SAC
Lavanda2.SACOS ::= Lavanda2.SAC
Lavanda2.SACO ::= #Lavanda1.Num
Lavanda2.LOTES ::= Lavanda2.LOT
Lavanda2.LOTES ::= Lavanda2.LOT
Lavanda2.LOTE ::= Lavanda1.TIPO
Lavanda1.TIPO ::= Lavanda1.CLASS
Lavanda1.CABEC ::= #Lavanda1.Da
Lavanda1.TINTO ::= br
Lavanda1.TINTO ::= cor
Lavanda1.FIO ::= alg
Lavanda1.FIO ::= la
Lavanda1.FIO ::= fib
Lavanda1.CLASSE ::= corpo
Lavanda1.CLASSE ::= casa
```

Fig. 4. Dependency graph in production Sacos

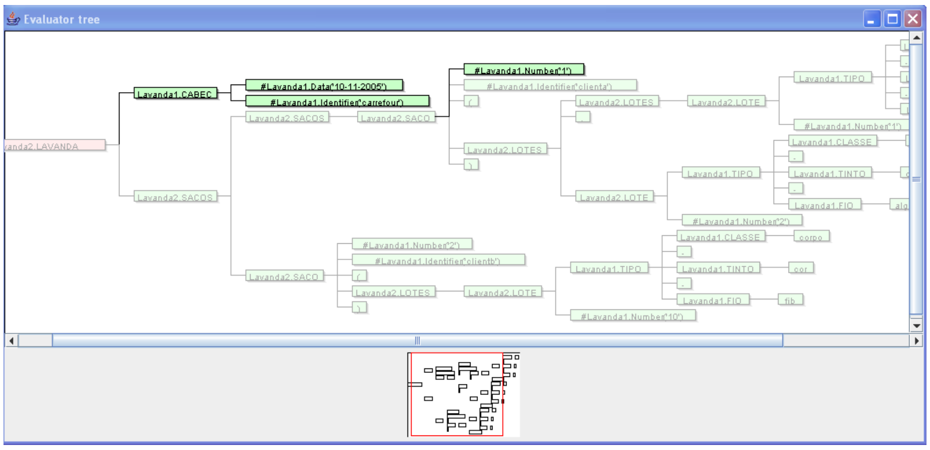


Fig. 5. Evaluator true

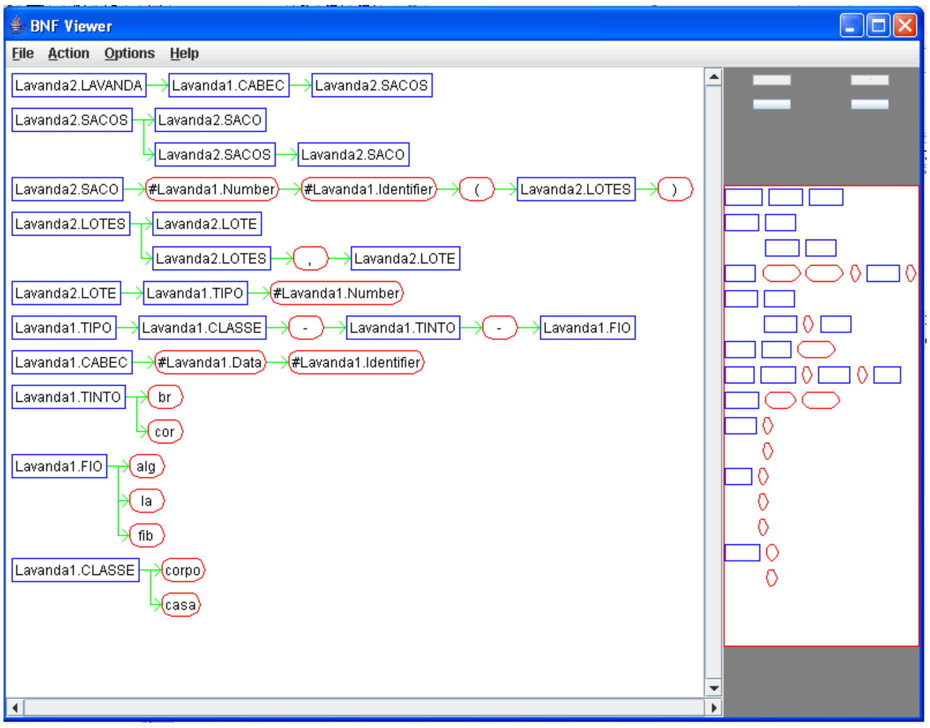
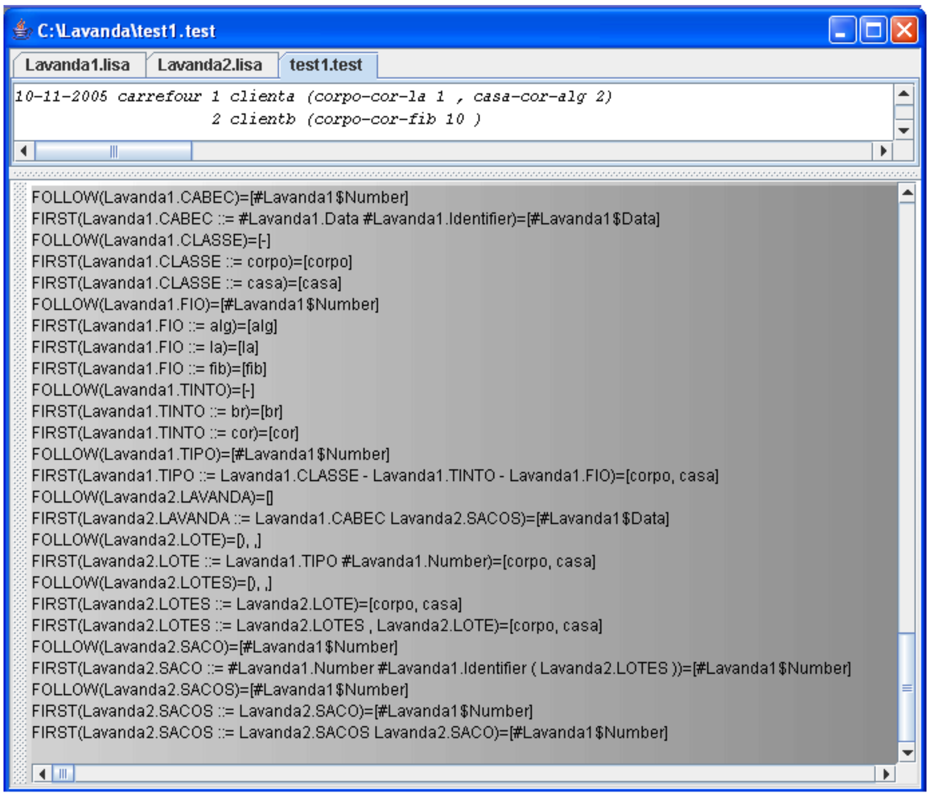


Fig. 6. BNF viewer



```
10-11-2005 carrefour 1 clienta (corpo-cor-la 1 , casa-cor-alg 2)
                     2 clientb (corpo-cor-fib 10 )
```

```
FOLLOW(Lavanda1.CABEC)=[#Lavanda1$Number]
FIRST(Lavanda1.CABEC ::= #Lavanda1.Data #Lavanda1.Identifier)=[#Lavanda1$Data]
FOLLOW(Lavanda1.CLASSE)=[-]
FIRST(Lavanda1.CLASSE ::= corpo)=[corpo]
FIRST(Lavanda1.CLASSE ::= casa)=[casa]
FOLLOW(Lavanda1.FIO)=[#Lavanda1$Number]
FIRST(Lavanda1.FIO ::= alg)=[alg]
FIRST(Lavanda1.FIO ::= la)=[la]
FIRST(Lavanda1.FIO ::= fib)=[fib]
FOLLOW(Lavanda1.TINTO)=[-]
FIRST(Lavanda1.TINTO ::= br)=[br]
FIRST(Lavanda1.TINTO ::= cor)=[cor]
FOLLOW(Lavanda1.TIPO)=[#Lavanda1$Number]
FIRST(Lavanda1.TIPO ::= Lavanda1.CLASSE - Lavanda1.TINTO - Lavanda1.FIO)=[corpo, casa]
FOLLOW(Lavanda2.LAVANDA)=[]
FIRST(Lavanda2.LAVANDA ::= Lavanda1.CABEC Lavanda2.SACOS)=[#Lavanda1$Data]
FOLLOW(Lavanda2.LOTE)=[), ,]
FIRST(Lavanda2.LOTE ::= Lavanda1.TIPO #Lavanda1.Number)=[corpo, casa]
FOLLOW(Lavanda2.LOTES)=[), ,]
FIRST(Lavanda2.LOTES ::= Lavanda2.LOTE)=[corpo, casa]
FIRST(Lavanda2.LOTES ::= Lavanda2.LOTES , Lavanda2.LOTE)=[corpo, casa]
FOLLOW(Lavanda2.SACO)=[#Lavanda1$Number]
FIRST(Lavanda2.SACO ::= #Lavanda1.Number #Lavanda1.Identifier ( Lavanda2.LOTES ))=[#Lavanda1$Number]
FOLLOW(Lavanda2.SACOS)=[#Lavanda1$Number]
FIRST(Lavanda2.SACOS ::= Lavanda2.SACO)=[#Lavanda1$Number]
FIRST(Lavanda2.SACOS ::= Lavanda2.SACOS Lavanda2.SACO)=[#Lavanda1$Number]
```

Fig. 7. First and Follow

6 AnTLR-3

AntLR (ANother Tool for Language Recognition) is a parser and translator generator tool that lets one define language grammars in either AntLR syntax (which is YACC and EBNF(Extended Backus-Naur Form) like) or a special AST(Abstract Syntax Tree) syntax. AntLR can create lexers, parsers and AST's.

6.1 A brief introduction to AnTLR-3

AntLR (formerly Purdue Compiler Construction Tool Set (PCCTS)) is a language tool that provides a framework for constructing recognizers, compilers, and translators from grammatical descriptions containing Java, C#, Python, or C++ actions. AntLR is popular because it is easy to understand, powerful, flexible, generates human-readable output, and comes with complete source. AntLR provides excellent support for tree construction, tree walking, and translation. There are currently over 5000 AntLR source downloads a month.

Terence Parr is the man behind AntLR and has been working on AntLR since 1989 [Par99,Par05]. He is a professor of computer science at the University of San Francisco. Together with his colleagues, Terence has made a number of fundamental contributions to parsing theory and language tool construction, leading to the resurgence of $LL(k)$-based recognition tools [PQ95,PQ96].

AnTLR-3 is a complete rewrite of the AntLR parser generator and is the culmination of over 15 years of experience building language tools. The software represents nearly four years of coding and research effort. From the user perspective, they are two primary new features: significantly enhanced parsing strength via $LL(*)$ parsing with arbitrary lookahead and vastly improved tree construction rewrite rules.

In practice, it means one can throw almost any grammar at AntLR that is non-left-recursive and unambiguous (same input can be matched by multiple rules); the cost is perhaps a tiny bit of backtracking, but with a DFA not a full parser. One can manually set the max lookahead k as an option for any decision though. The $LL(*)$ algorithm starts to use more lookahead when it needs to and is much more efficient than normal LL backtracking. Lexers are much easier due to the $LL(*)$ algorithm as well.

6.2 Lavanda implementation

Using the DSL from section 3, we wrote an AnTLR-3 implementation. Due to space limitations, we have chosen to present only two productions.

```
sacos returns [int nSacos = 0, HashMap env = new HashMap()]
@init { LinkedList clientIds = new LinkedList();
        LinkedList bagIds    = new LinkedList(); }
        :       ( saco[$env, clientIds, bagIds] { $nSacos++; } )+
        ;
```

Attributes: This production only has two synthesized attributes — *nSacos* and *env*. The synthesized attributes are declared using the `returns` construction. Later one can refer to them using the `$attr` construction.

Initialization actions: The construction `@init{..}` allows one to write initialization code for a rule statement. The code is written in the target programming language.

Passing attributes: To pass attributes to other productions, AnTLR-3 uses a similar approach to function parameter passing. After the production rule, one just put all the attributes inside the `[..]` guards.

Semantic actions: AnTLR-3 allows one to put semantic actions anywhere on the rule statement. These actions are identified by the `{..}` construction and must be written in the target programming language.

EBNF notation: AnTLR-3 uses the EBNF notation on the rule declaration. This allows the use of regexp repetition like operators such as `*`, `+` and `?`. This form helps writing recursive productions without the hard job of rewriting a left recursive grammar (compare this implementation with the YACC one on section 4).

```
lotes [ HashMap inEnv ] returns [ HashMap outEnv, int nLotes = 0, int custoTotal = 0]
    :        l1=lote[$inEnv]  { $nLotes++; $outEnv = $l1.outEnv; }
        (',' l2=lote[$outEnv] { $nLotes++; $outEnv = $l2.outEnv; } )*
    ;

lote [ HashMap inEnv ] returns [ int custoTotal, HashMap outEnv ]
    :   tipo NUM {
                  $inEnv.put($tipo.name, (Integer) $inEnv.get($tipo.name) + 1);
                  $outEnv = $inEnv;
                  }
    ;
```

This second example shows an inherited attribute *env*, and again the EBNF construction that helps writing a non left-recursive grammar.

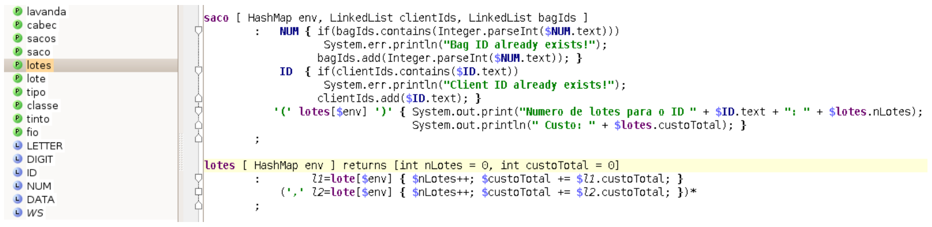
6.3 Generated Tools

Along with the CG, AnTLR-3 offers a number of useful language-based tools designed to help the grammar construction and analysis. One of this tools is the *ANTLRWorks: The ANTLR GUI Development Environment*[3].

Grammar edition ANTLRWorks provides an excellent grammar editing environment as developers have come to expect. It is available as a standalone tool or a plugin. The editor supports syntax highlighting, rule navigation tree, auto-indentation, refactoring, sensitive auto-completion and a long list of other features.

You can see a screenshot of the main environment on ANTLRWorks on figure 8.

[3] `http://www.antlr.org/works/index.html`



```
saco [ HashMap env, LinkedList clientIds, LinkedList bagIds ]
        :   NUM { if(bagIds.contains(Integer.parseInt($NUM.text)))
                   System.err.println("Bag ID already exists!");
                  bagIds.add(Integer.parseInt($NUM.text)); }
            ID  { if(clientIds.contains($ID.text))
                   System.err.println("Client ID already exists!");
                  clientIds.add($ID.text); }
           '(' lotes[$env] ')' { System.out.print("Numero de lotes para o ID " + $ID.text + ": " + $lotes.nLotes);
                                 System.out.println(" Custo: " + $lotes.custoTotal); }
        ;

lotes [ HashMap env ] returns [int nLotes = 0, int custoTotal = 0]
        :       l1=lote[$env] { $nLotes++; $custoTotal += $l1.custoTotal; }
            (',' l2=lote[$env] { $nLotes++; $custoTotal += $l2.custoTotal; })*
        ;
```

Fig. 8. ANTLRWorks editor environment

This language-based tool has the capability to show the decision DFA of any production, as well as the rule dependency graph. These diagrams can be exported easily to a bitmap image or a EPS file. When there are conflicts on the grammar, ANTLRWorks can display nondeterminism warnings as ambiguous paths through the syntax diagram. Some examples of these features are show on figures 9, 10, 11 and 12.

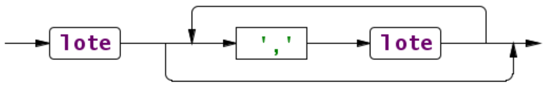


Fig. 9. ANTLRWorks syntax diagram for production "lotes"

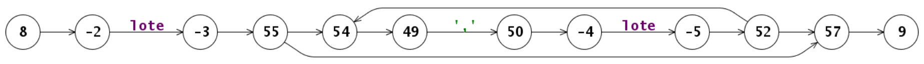


Fig. 10. ANTLRWorks decision DFA for production "lotes"

Grammar depuration Because of AnTLR-3's new functionality, ANTLRWorks can immediately interpret a grammar and test it against some sample input–without generating anything! This functionality is great for rapid prototyping. One can get the parse tree as a result of interpreting a grammar. Imagine passing some input to a rule in your grammar and instantly seeing how the rule matches the input. An example can be found on figure 13.

AnTLR-3 has a sophisticated debug event mechanism that allows ANTLR-Works to follow along as a parser processes input. AnTLR-3 includes a well-defined protocol for communicating with remote parsers so ANTLRWorks can

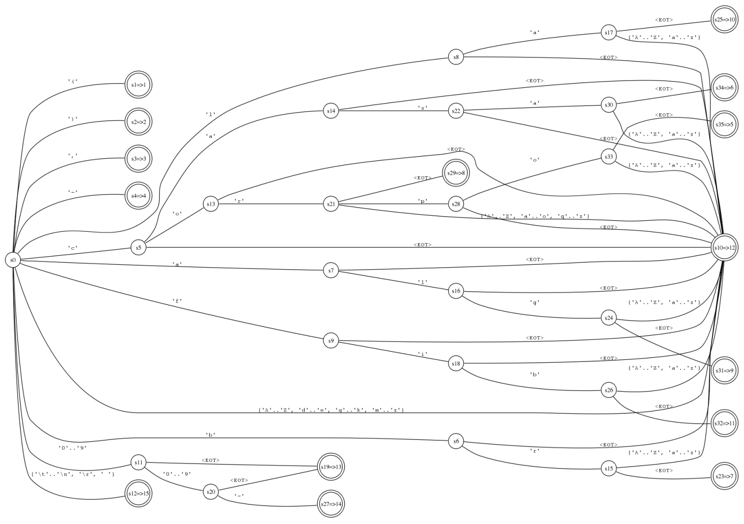


Fig. 11. ANTLRWorks lexer decision DFA

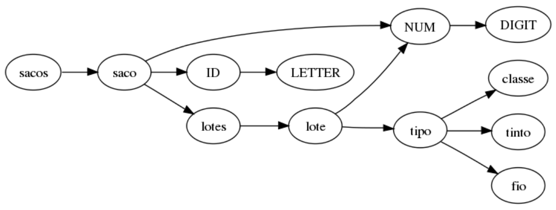
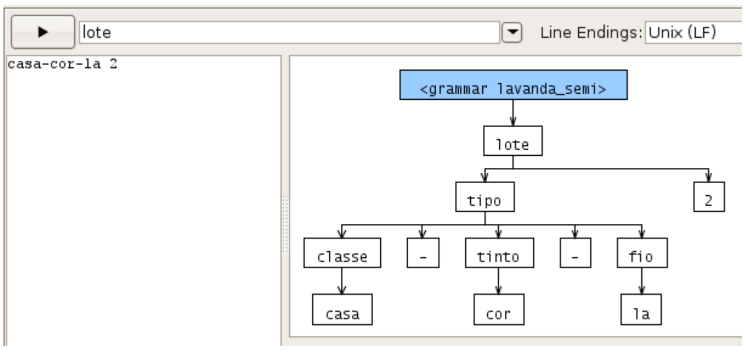


Fig. 12. ANTLRWorks rule dependency graph

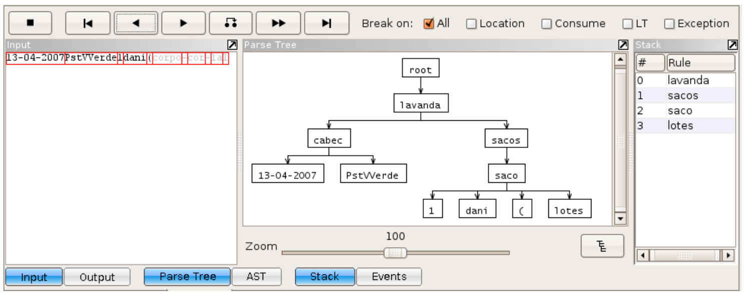

lote

Line Endings: Unix (LF)

```
casa-cor-la 2
```

Fig. 13. ANTLRWorks parse tree for a recognition of the “lote” production

actually connect to a parser generated in any language with a socket library. The debugger was designed so one can can pause a running parser and then rewind it! Once a parse has completed, ANTLRWorks has a complete trace and allows the user to walk back and forth over the input stream like a video camera. An example debug session can be seen on figure 14.



13-04-2007PstVVerde1dani(corpo-cor-1a1

#	Rule
0	lavanda
1	sacos
2	saco
3	lotes

Fig. 14. ANTLRWorks paused debug session

For those programmers generating Java, ANTLRWorks knows how to generate Java, build a test harness, compile everything, launch the parser, and connect to it — a great rapid prototyping feature when one can't use the interpreter because actions and/or semantic predicates must execute.

7 Conclusion

In order to get a feeling on how much the choice of the Compiler Generator influences a language implementation project, we decided to define a DSL (Lavanda) and develop a language processor for it with different generators.

In this paper we reported the work carried on to measure the effort done in each case, the benefits obtained, and the readability of each specification.

The implementation of Lavanda Language with the three tools (experiments described along the paper) allowed us to summarize the most important characteristics of each one, and to conclude that YACC is the poorest and in some sense out of date, while the other two, LISA and AnTLR-3, are very similar and effectively good tools according to the three vectors used for the comparison study—usability, complementary aids, and quality of the final product.

Table 2 shows that synthesis; rows represent evaluation parameters and columns correspond to the CG tested. Rows are divided in three sets of parameters: the first four parameters are related with *language specification*; the next six parameters are related with *generated compiler features*; and the last three are related with the *generator*.

In the next paragraph we present some of the reasons that support our choice of the criteria above.

	YACC	LISA	AnTLR-3
Generation of Lexer and Parser	Parser only	Both	Both
Input Metalanguage	BNF	BNF	EBNF
Language Spec. Formalism	TG	AG	TG or AG
Semantic Spec. Language	C	Java	Target Language
Kind of Parser	BU ($LALR(1)$)	BU ($LL(1)$, $LALR(1)$)	TD ($LL(*)$)
Target Languages	C	Java	Java,C#, C++, Python
Error Handling	Rudimentar	Java Exceptions	Target Exceptions
Standalone Compiler	Yes	Yes	Yes
Generation of Editors	No	Yes	Yes
Compiler Debugging Features	No	Yes	Yes
Development Environment	No	Yes	Yes
Graphical Interface	No	Yes	No
Grammar Analyzes Tools	+/-	Yes	Yes

Table 2. Generator feature matrix

The language specification can be specified using several formalisms like attribute grammars or translation grammars. These formalisms allows us to define the syntax and the semantics of the language, using a special notation to write the grammar productions (like BNF or Extended BNF) and a standard language (like C or Java) to specify the semantic actions associated to each production. In some cases, the CG generates only the parser but in another cases it also produces the lexer (in those cases the lexical specification complements the syntactic description). We define four parameters for the language specification criterion: inclusion of lexer generation; metalanguage supported (BNF, EBNF, other); semantics specification by a translation or an attribute grammar; language to specify the semantic rules.

The compiler generated can be based on different kinds of parsers, and incorporate different techniques to detect lexical, syntactic and semantic errors. Also it can be written in different languages. The generated compiler can run independently of the CG or it can require the presence of CG libraries. Nowadays, it is also possible to generate language based tools that enhance the compiler, like structured editors, visualizers and animators. The parameters that we use in this second criterion are: Kind of Parser, Target Languages, Error Handling, Stand-alone Compiler, Generation of Editors and Compiler Debugging Features.

At last, the third criterion is related with the CG. In order to support the compiler construction process, some tools offer a friendly user interface. This interface usually includes graphical features and may be some tools to aid analyzing the grammar. The parameters used here are: Development Environment, Graphical Interface and Grammar Analyzes Tools.

To choose a generator, looking to the comparison table 2, the developer should be aware of the following hints: an attribute grammar is a more complete formalism; bottom-up parser is more powerful, although a LL grammar is more natural; error handling mechanism is an important benefit of the compiler; the target language influences the compiler efficiency.

References

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques and Tools*. Addison-Wesley, 1986.

[dCH06] Daniela da Cruz and Pedro Rangel Henriques. Lavanda, an exercise with attribute grammars and a case-study to compare ag-based compiler-generators. Cctc technical report, Dep.Informática / Univ. do Minho, Dec. 2006.

[HU79] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Reading, MA, 1979.

[HVM$^+$05] Pedro Henriques, Maria João Varanda, Marjan Mernik, Mitja Lenic, Jeff Gray, and Hui Wu. Automatic generation of language-based tools using lisa system. *IEE Software Journal*, 152(2):54–70, April 2005.

[HVML02] Pedro Henriques, Maria João Varanda, Marjan Mernik, and Mitja Lenic. Automatic generation of language-based tools. In *LDTA - Workshop on Language, Descriptions, Tools and Applications (ETAPS'02)*, April 2002.

[Joh75] Stephen C. Johnson. YACC yet another compiler compiler. Computing Science Technical Report CSTR32, Bell Laboratories – Murray Hill, New Jersey, 1975.

[Knu68] Donald Knuth. Semantics of contex-free languages. *Math. Syst. Theory*, 2(2):127–145, 1968.

[LMB92] J.R. Levine, T. Mason, and D. Brown. *Lex & Yacc*. Ed. Dale Dougherty. O'Reilly & Associates Inc., 1992.

[LS75] M. E. Lesk and E. Schmidt. Lex - a lexical analyzer generator. Computing Science Technical Report 39, Bell Laboratories – Murray hill, New Jersey, 1975.

[MKŽ95] Marjan Mernik, Nikolaj Korbar, and Viljem Žumer. LISA: A tool for automatic language implementation. *ACM SIGPLAN Notices*, 30(4):71–79, Abril 1995.

[MLAŽ98] Marjan Mernik, Mitja Lenič, Enis Avdičaušević, and Viljem Žumer. A reusable object-oriented approach to formal specifications of programming languages. *L'Objet*, 4(3):273–306, 1998.

[MLAZ00] Marjan Mernik, Mitja Lenic, Enis Avdicausevic, and Viljem Zumer. Compiler/interpreter generator system LISA. In *IEEE Proceedings of 33rd Hawaii International Conference on System Sciences*, 2000.

[MLAŽ02] Marjan Mernik, Mitja Lenič, Enis Avdičaušević, and Viljem Žumer. LISA: An Interactive Environment for Programming Language Development. In Nigel Horspool, editor, *11th International Conference on Compiler Construction*, volume 2304, pages 1–4. Lecture Notes in Computer Science, Springer-Verlag, 2002.

[MNA$^+$01] M. Mernik, U. Novak, E. Avdičaušević, M. Lenič, and V. Žumer. Design and implementation of simple object description language. In *ACM Symposium on Applied Computing, SAC'2001*, pages 590–594, 2001.

[MZLA99] Marjan Mernik, Viljem Zumer, Mitja Lenic, and Enis Avdicausevic. Implementation of multiple attribute grammar inheritance in the tool lisa. *ACM SIGPLAN not.*, 34(6):68–75, Jun. 1999.

[Paa95] Jukka Paakki. Attribute grammar paradigms - a high-level methodology in language implementation. *ACM Computing Surveys*, 27(2):196–255, 1995.

[Par99] Terence Parr. Practical computer language recognition and translation – a guide for building source-to-source translators with antlr and java. http://www.antlr.org/book/index.html, 1999.

[Par05] Terence Parr. An introduction to antlr. http://www.cs.usfca.edu/ parrt/course/652/lectures/antlr.html, Jun. 2005.

[PQ95] Terence Parr and Russell W. Quong. Antlr: A predicated-ll(k) parser generator. *Software Practice and Experience*, 25(7):789–810, July 1995.

[PQ96] Terence Parr and Russell W. Quong. Ll and lr translator need k. *SIGPLAN Notices*, 31(2), Feb. 1996.

[Rep82] Thomas Reps. *Generating Language-Based Environments*. PhD thesis, Cornell University, 1982.

[WG84] William Waite and Gerhard Goos. *Compiler Construction*. Texts and Monographs in Computer Science. Springer-Verlag, 1984.