DSL Contest - Evaluation and Benchmarking of DSL Technologies

Kim David Hagedorn, Kamil Erhard
Advisor: Tom Dinkelaker

September 30, 2009

Contents

Chapter 1

Introduction

A Domain Specific Language (DSL) is a formal language specified to solve problems of certain domains. DSLs aim at solving problems faster and easier than general-purpose languages, but these improvements are traded off by having to implement a separate DSL for every problem domain. Thereby arises the question, how fast and easy new DSLs can be specified and applied to a problem. Our framework compares several tools to implement DSLs and measures ease of development and efficiency of the generated tools.

In our model problem we generate tools to read and execute state machines and test their execution speed by recognizing words of large texts. We compared six different approaches, of which five are tools to specify a DSL and one is a pre-existing DSL for state machines (Apache SCXML). We also developed a framework to compare DSLs, measure development and execution parameters and automatically generate documents from the test results.

1.1 Domain Specific Languages

Domain Specific Languages are formal languages tailored to specific problem domains. These problems usually involve computational tasks, so in order to use a specified DSL one has to provide tools to automatically parse a text in that DSL to a datamodel and perform computational tasks on the defined model.

It is possible to manually generate such a parser, but is generally not recommended, as the process of parsing a linear text to a multi-dimensional datamodel is rather complicated and generating a program to do so is a complicated, mechanical and error-prone problem. This is why tools for DSLs usually contain a *parser generator*, that takes a formal definition of a language, also known as *grammar*, and generates a *parser*, which is a program to read a text and generate a datamodel from it.

Some parsers are split into the actual parser and a separate *lexer* or *scanner*, which recognizes single tokens such as words and parentheses and passes them to the parser. These approaches are usually faster, but less flexible, as some

keywords of the language can not be used otherwise, for example as variable names. If a scanner is present in a DSL tool it might be necessary to provide a table of possible tokens and their meaning, but some tools can also generate these tables.

The generated datamodel is usually a *syntax tree*, which contains nodes for every element of the document and elements that belong to a larger entity, such as words belonging to a sentence, are grouped as children under that entity. A developer then has to implement a programm to perform computational task on this syntax tree. Albeit this is always a manual process, the choosen DSL tool plays a role as some are restricted in the programming language of the produced syntax tree. Antlr for example will always generate Java objects and Bison will usually generate c structs.

1.2 Measuring DSL tools

In the previous section some design choices in the implementation of DSL tools were presented. To see, how these design choices and further aspects of individual DSL tools affect ease of implementation and execution speed of generated tools, this framework measures some parameters regarding development of a DSL using a certain tool as well as application of the generated tools to a reference problem. These parameters are:

Development

- **Development time:** How much time is needed to implement the reference language using this technology.
- **Description of development:** How did we precede in the implementation. Which tools were used to generate which artifacts. Which problems did we occur?
- **Lines of Code:** How much lines of code needed to be written.
- **Syntactic noise:** How exactly can this tool match the reference language. We measure the textual difference between the reference language and the language that this tool can process, using the *Levenshtein*-distance [7].

Execution

- **Execution speed:** How much time do the generated tools need to process problems, depending on different input sizes.
- **Correctness:** Does the produced output match the expected output.
- **Memory usage:** How much memory did the tools need over time, how often did Java solutions invoke the garbage collection.

In the following chapter we present our framework that measures these parameters and generates a readable document from the result.

Chapter 2

The DSL Contest Framework

2.0.1 Implementation

The DSL Contest Framework is mainly an integrated set of ant build script. It is designed to support and benchmark DSL technologies implemented in any computer language. To date we have included six different DSL technologies in our framework. Four out of these six technologies generate runnable JAVA code, the other two generate C code. The DSL approaches will be discussed later on in detail. Every one of the DSL approaches are located in a sub directory under the root directory of the DSL Contest framework. Every sub directory contains an ant build script that represents the main control of flow.

We decided to use a state machine representation as the example DSL for reviewing technologies. This DSL specification should be self explainable.

Listing 2.1: The examined DSL specification

```
machine Watch {
        start state reseted {
                entry: output "resetTimer";
                transitions {
                        when start enter running;
                        when switchOff enter off;
                }
        }

        state running {
                entry: output "startTimer";
                perform: output "loopRunning";
                transitions {
                        when split enter paused;
                        when stop enter stopped;
                }
        }

        state paused {
                entry: output "pauseTimer";
                transitions {
```

```
                        when unsplit enter running;
                        when stop enter stopped;
                }
        }

        state stopped {
                entry: output "stopTimer";
                transitions {
                        when reset enter reseted;
                        when switchOff enter off;
                }
        }

        state off {
                exit: output "switchOff";
                transitions {
                        when toEnd end;
                }
        }
}
```

To this day we have implemented a couple of benchmark and data mining tasks that are executed by the respective sub ant build file.

The benchmark tasks deal with the following situation. A concrete state machine gets generated and is supplied together with a file of events, representing transition events. The state machine starts in its specified start state. An event producer reads events line for line and sends them as input to the concrete state machine. The concrete state machine executes these transitions and changes its state. If the state machine enters an end state, it terminates.

For the benchmark and data mining tasks we generate a concrete state machine capable of recognising input texts. The input text is simultaneously transformed into a line for line list of events. Each event represents one character of the input text. The example input text is a 512 KiB character text.

In the following we enumerate the targets of the build scripts, explain the semantic meaning and state in general how they are implemented.

build

The build target builds any necessary code needed by the benchmark and data mining tasks. In general this will build parser and lexer code for the given DSL specification.

clean

The clean target just cleans any generated files, therefore restoring the original file and directory set.

benchmark

The benchmark target is just an alias for running all other benchmark and data mining targets, enumerated hereafter.

runtime

The runtime target runs the concrete state machine with the full input text and measures the runtime. The following data is written in the sub directory:

- **benchmark-results/N/time**: The total time measured
- **benchmark-results/N/machine.sm**: The used concrete state machine
- **benchmark-results/N/machine.events**: The used file of events
- **benchmark-results/N/machine.expectedOutput**: The expected state transitions
- **benchmark-results/N/output**: The actual state transitions
- **benchmark-results/N/num-states**: The size of the input text used for the concrete state machine

runtimes

The runtimes target calls the runtime target a specified number of times (10). The first runtime call will use only one chunk of the full input text. The second call will use two chunks and so on. The Nth (10th) call is identical to a regular runtime call. The data is written in the directory benchmark-results/i/, where i stands for the ith iteration.

space

The space target tries to measure the used space of the DSL technology. Currently it only supports the measurement of JAVA technologies and measures heap size and garbage collection statistics.

- **benchmark-results/space/total-time**: The total time measured
- **benchmark-results/space/gc-time**: The time used by the garbage collection
- **benchmark-results/space/gc-counts**: The number of times the garbage collection was executed
- **benchmark-results/space/heap-Usage**: The usage of the heap size during 100ms intervals
- **benchmark-results/space/max-heap-size**: The maximal size of heap used
- **benchmark-results/space/gc-percentage**: The fraction of gc-time and total-time

loc

The loc target counts the number of lines and characters of given files. It counts separately generated and user written files. Furthermore it tries to count effective lines and characters of code in both cased, by trimming whitespaces and removing comments.

- **benchmark-results/loc/lines-generated**: The number of generated lines of code
- **benchmark-results/loc/elines-generated**: The number of generated effective lines of code
- **benchmark-results/loc/lines-userwritten**: The number of user written lines of code
- **benchmark-results/loc/elines-userwritten**: The number of user written effective lines of code
- **benchmark-results/loc/chars-generated**: The number of generated characters of code
- **benchmark-results/loc/echars-generated**: The number of generated effective characters of code
- **benchmark-results/loc/chars-userwritten**: The number of user written characters of code
- **benchmark-results/loc/echars-userwritten:** The number of user written effective characters of code
- **benchmark-results/loc/lines-total**: The number of total lines of code
- **benchmark-results/loc/elines-total**: The number of total effective lines of code
- **benchmark-results/loc/chars-total**: The number of total characters of code
- **benchmark-results/loc/echars-total**: The number of total effective characters of code

validity

The validity target just validates that the output produced by a concrete state machine is equal to the expected output. In this case it checks whether the state machine performs the right state transitions.

- **benchmark-results/N/valid**: true if benchmark-results/N/machine.expectedOutput is equal to benchmark-results/N/output. Otherwise false.

environment

The environment target gathers some interesting information about the environment runtime. Currently it prints some information about the used JAVA Virtual Machine version and the operating system.

- **benchmark-results/java-version**: The JAVA version used by ant
- **benchmark-results/java-vm**: The JVM used by ant
- **benchmark-results/timestamp**: The time of the benchmark execution

syntactic-noise

The syntactic-noise target calculates the syntactic noise between the reference DSL specification and the one used by the DSL technologies. Currently it gets calculated with the Levenshtein distance [7].

- **benchmark-results/levenshtein-distance**: The calculated levenshtein-distance

documentation

The documentation target integrates the results and outputs of the benchmark and data-mining targets and creates a Latex document out of it. Therefore it gets called after the benchmark target.

Individual results are named results*.inc.tex, where * is a wild card for any character. All found results*.inc.tex files will be concatenated. The concatenated results document will be included in this document's Benchmark Results/Results by Technology section.

Comparisons with other technologies are named comparisions*.inc.tex and will also be concatenated. The concatenated comparisons document will be included in this document's Benchmark Results/Comparisons section.

2.0.2 Extensibility

The DSL contest is designed to be extended both with new DSL technologies for review and new benchmark and data mining tasks. In the following section we show a step by step introduction for adding a new DSL technology. The antlr technology is considered as our reference implementation.

Example: Adding a new Technology

1. Create a new sub directory under the root directory named after the DSL technology
2. Generate the necessary parser, lexer and runtime code for the supplied reference DSL specification. The runtime code should take two arguments: the location of the concrete state machine and the location of the list of events for the concrete state machine, as explained earlier
3. Create a build.xml file in the sub directory with the targets explained earlier or copy the skeleton-build.xml file located in the root directory
4. Modify the build target to generate the runtime code including the parser
5. Write the other needed targets either by using the help targets in common.xml with the respected arguments or by implementing it yourself. Assure that you output the results in the locations specified earlier.
6. Create a directory doc and create a file called "name" with the name of the DSL technology. Furthermore append a short description about the technology and the implementation process in a file called "description.inc.tex".
7. Set up the projects you want to benchmark and document by modifying the fileset pattern in the root build.xml file in the DSL Contest folder. For example exclude some technologies by using the exclude attribute.

8. Now you should be able to call the benchmark and build-pdf targets of the root build.xml file in the DSL Contest folder.

Chapter 3

Examined DSL Tools

3.0.3 Apache SCXML

Description

Apache SCXML [1] is not a tool to generate DSL but an example for a pre-existing DSL for state machines.

SCXML is a XML representation of a state machine. Apache provides the SCXML Commons, which are a set of Java utilities to parse and execute SCXML files to a Java execution model. A developer can then send events to this state machine or register listeners to get notified of state changes.

Usage in DSL Contest

In this case there was no grammar or parser that needed to be implemented, all that was needed was a generator to create state machine definitions in the SCXML format (`SCXMLGenerator.class`) and a `SCXMLRunner.class` that parses a file to a state machine using the `SCXMLParser.parse`-method.

To do so it was necessary to implement two different error reporting classes (`ErrorHandler`, `ErrorReporter`) and a `Evaluator` class that - in this case - is only responsible for creating new `Context` Objects. The printing of output messages by adding a custom tag to SCXML did not work, so a workaround was implemented as a Listener (`PrintStateListener`), that simply prints a state's name at every state change.

Development Experience

Integrating SCXML into our project took approx. five hours, of which one was dedicated to develop a custom generator that produces correct SCXML state machines and four hours to implement a class that parses a state machine, feeds it with events and prints the output to the desired location. Two

of these four hours were dedicated to fix a bug that prevented Apaches XML parser (`XMLDigester`) from correctly recognizing custom SCXML tags. Since this problem seemed not to be resolvable, a workaround was implemented that simply prints a state's name on transitions.

3.0.4 Stratego/XT

Description

Stratego [5] and the XT transformation tools are a collection of tools to represent data structures as terms and to manipulate them using formal rules.

They use *ATerms* (annotated terms) as formal representations of graphs. A term has a name (*Constructor*) followed by a set of subterms as content in parentheses. A text can be parsed to a structured ATerm using the *SDF tools*. These tools need a *parse table* that equates to a grammar definition. Such an ATerm can then be modified using *rewrite rules*. A rule has an ATerm as left-hand side that represents the current term and a right-hand side that represents the term after rewriting. A rule is made abstract by introducing variables for subterms.

Several rules can be composed to a strategy, that declares the sequence of rules and possible traversing of the term. An ATerm can finally be written back to a plain text files using additional rewrite rules.

Usage in DSL Contest

This implementation parses a state machine definition to an ATerm, rewrites it to an ATerm representation of a Java source file and translates this term to a Java program representing a concrete state machine.

The `Lex.sdf` file defines individual tokens and the The `StateMachine.sdf` defines their composition to an ATerm representation of a state machine. These files are composed to a parser table for the `sglri`-tool that parses state machine files to aterms. The `aterm2java.str`-strategy defines rules to translate elements such as states and transitions to ATerm representations of Java code fragments and composes them to a *Compilation Unit*, a representation of a complete class file. This strategy is compiled to a c program using the `strc`-command. The output of the resulting aterm2java-program is then fed to `pp-java`, a tool that converts ATerm representations of Java source files to pretty-printed Java sourcecode. This Java source code is then compiled to a class that represents the defined state machine.

The created Java class is a subclass of `AbstractStateMachine`, an abstract state machine model that was written for this approach. States can be added to this machine using inlined subclass declarations of the abstract class `State` and binding them to a class variable name. This was necessary to avoid the Java method size limit of 64K per method. The `AbstractStateMachine` class

also provides the execution model to run event files on a StateMachine.

Development Experience

Development of a state machine DSL using Stratego/XT took 12 hours in total. Defining a grammar to read a state machine definition to an ATerm was straightforward and quickly accomplished. Term rewriting and strategies in general were also no complicated task, but creating an ATerm representation of a Java program (adhering to the syntax understood by `pp-java`) was complicated and time consuming. The *java-tools* provided by the Stratego/XT website allow direct inlining of java code in the right-hand side of rewrite rules, but this approach did not work here for some ugly masking/unmasking issues. Therefore all elements of the state machine had to be translated to large ATerm representations of classes, methods or variables.

3.0.5 AntLR

Description

Source: http://www.antlr.org

> ANTLR, ANother Tool for Language Recognition, is a language tool that provides a framework for constructing recognizers, compilers, and translators from grammatical descriptions containing actions in a variety of target languages. ANTLR automates the construction of language recognizers. From a formal grammar, ANTLR generates a program that determines whether sentences conform to that language. In other words, it's a program that writes other programs. By adding code snippets to the grammar, the recognizer becomes a translator or interpreter. ANTLR provides excellent support for intermediate-form tree construction, tree walking, translation and provides sophisticated automatic error recovery and reporting. Text from the AntLR homepage.

Usage in DSL Contest

AntLR was used to generate a parser and lexer for the given DSL specification. We created a file containing the grammar in Extended BackusNaur Form (grammar/Statemachine.g) This file includes also the Tokens for the lexer.

AntLR stand alone does not generate concrete model code, but rather builds an abstract syntax tree. Therefore we wrote a simple state machine (src/antlr/fsm) ourselves and inlined java instructions directly in the grammar file and let AntLR build our state machine. By using Hashtables for the transitions and states we assured a pretty fast implementation.

Running AntLR on the grammar file resulted in the parser (src/antlr/generated/StateMachineParser.java) and lexer (src/antlr/generated/StateMachineLexer.java).

The runtime class src/antlr/run/Runner.java uses the parser and lexer to read a given input state machine in the specification format and instantiates a concrete state machine. This concrete state machine can be used in the benchmark tests together with a given events file.

Development Experience

Developing in AntLR was pretty easy. The software is well documented and and writing the grammar in EBNF is easy. The lack of model code generation is compensated by allowing the inclusion of java code directly in the grammar file. This results in a very powerful but rather low level code. Splitting of grammar files allows a modularised approach.

The development took a short time. Approximately three hours were used to the grammar file. Five Hours were used to write the semantics.

3.0.6 Bison+Flex

Description

GNU bison [3] is a open source implementation of the UNIX program yacc and generates a parser from a context-free grammar in Backus-Naur-Form. As bison does only generate a syntactical analyzer, it needs a separate lexical analyzer (scanner). This implementation uses the flex [2] scanner generator to perform this task.

Usage in DSL Contest

Every token in a bison grammar is associated with a c type, so all elements of the abstract syntax tree have to be provided as a c header file (*statemachine.h*) that provides the needed types as structs. The rules are defined in a grammar file (*statemachine.y*). Every rule is associated with some lines of c code that set up the neccesary data structures and associate the rule with a return value ($$), using the tokens of the rules right-hand side ($1 ... $i). bison generates a c program containing an yyparse() method that parses texts in the defined language to c structs.

A bison parser needs a scanner like flex to identify terminals and associate them with types. How tokens are identified and associated with types is done by providing a scanner definition (*statemachine.l*) that defines every token as a regular expression and associates it with a c code snippet that returns an integer value indicating the token's type. flex generates a c program providing an yylex() method, that identifies tokens and can be used by the parser.

An execution model for the state machine had to be implemented by hand (*statemachine.c*). The generated program invokes the yyparse()-method to generate a struct-representation of an input file and follows transitions specified by an event file. It uses hashtables (*gHashTable*) to look up states and transitions and saves a pointer to the following state on the first visit of a transition.

Development Experience

Implementing the state machine language with flex and bison took approximately 25 hours. About 7 hours were needed to understand the structure of bison and flex and to generate a scanner and a parser. The rest of the time was needed to implement a representation of a statemachine as c structs and to implement an execution enginge for state machines. A large part of that time (approx. 10h) was needed to debug the creation of structs and of the engine, which is a time-consuming task with c programs. These bugs were mainly segmentation faults. Since bison will reuse its internal variables ($$, $1 .. $i) and will not allocate memory for each token, a programmer has to dynamically allocate memory for every token that he wants to use after parsing.

3.0.7 DSL2JDT

Description

> The Eclipse Java Development Tools (JDT) excels at supporting the editing and navigation of Java code, setting the bar for newer IDEs, including those for Domain Specific Languages (DSLs). Although IDE generation keeps making progress, most developers still rely on traditional ways to encapsulate new language abstractions: frameworks and XML dialects. We explore an alternative path, Internal DSLs, by automating the generation of the required APIs from Ecore models describing the abstract syntax of the DSLs in question.
>
> Most embedded DSLs, while offering a user-friendly syntax, are fragile in the sense that the IDE does not flag the embedded statements that break the well-formedness rules of the DSL (because the IDE checks only the well-formedness of the host language, usually Java). To address this issue, we leverage the extension capability of Eclipse to detect at compile-time malformed DSLs expressions. The technique relies on mainstream components only: EMF, OCL, and JDT.

Usage in DSL Contest

DSL2JDT [6] was used in conjunction with EMF and Ecore. While EMF and Ecore generated the model code, DSL2JDT generated the Parser. We started by writing an Ecore description of our DSL specification, resulting in a XML representation of an Entity-Relationship-Modell (ecore/statemachine.ecore). EMF was then used to built model code (src/dsl2jdt/generated) out of the ecore model. Finally DSL2JDT generated a parser (src/dsl2jdt/generated/StateMachineExprBuilder.java).

All these steps were done in the Eclipse IDE. Building outside of Eclipse seems to be impossible, because all of these tools are plugins for Eclipse, therefore an integration in our ant build system was not possible.

The generated parser accepts input in form of instantiated JAVA objects. During developing this makes perfect sense, but has a serious limitation. JAVA classes have a size limit. In our case during benchmarking we exceed this limit. Therefore we wrote a simple transformer (smGen/DSL2JDTLetterTree.java), that reads a state machine specification, creates an object iteratively out of it and serialises it. Later on the serialised object can be used for benchmarking. We defined the semantics of the state machine in src/dsl2jdt/run/StateMachineRunner.java. This class takes the serialised object and an event file as input, starts the machine and executes the transitions, specified by the events.

The generated model code uses collections with linear lookup time for the states and transitions, although we specified maps and keys in the ecore model. This would significantly increase the runtime speed, therefore we used a custom hashtable as a cache for looked up states and transitions. The cache can be switched off by an command line argument for better comparison.

Development Experience

The creation of the ecore model was pretty easy. It took some time to realise, that EMF would always use collections with linear lookup time, instead of maps with key/value pairs. The usage of fluid interfaces looks pretty good and makes it very elegant to write a DSL with it. Although we have not really tested it, the integrated support of a powerful IDE like eclipse should be very comfortable.

Approximately two hours were used to write and validate the ecore model. The semantics (StateMachineRunner) required another three hours. The Building of the mentioned transformer consumed another two hours.

3.0.8 Monticore

Description

> Monticore [4] is a framework for an efficient development of domain-specific languages (DSLs). It processes an extended grammar format which defines the DSL and generates components for processing the documents written in the DSL. Examples for these components are parser, AST classes, symboltables or pretty printers.This enables a user to rapidly define a language and use it together with the MontiCore-framework to build domain specific tools.

Usage in DSL Contest

We used Monticore to generate both parser and lexer and furthermore model code. Monticore uses internally AntLR for parser and lexer generation but extends them with its needed functionality. Monticore is a client/server application. The parsing of the grammar file is done on client side, further processing is done on the Monticore server. We wrote a Monticore grammar file for our DSL specification (grammar/def/monticore/generated/StateMachine.mc). The syntax is written in EBNF and therefore similar to the one of antlr.

Running Monticore on the grammar requires an user account on Monticore's website. It then generates model code (src/monticore/generated/_ast) and the parser and lexer (src/monticore/generated/_parser).

We defined the semantics of the state machine in src/monticore/run/StateMachineRunner.java. This class takes a state machine in the specification form and an events file, starts the state machine and executes the transitions according to the events file.

The specification file had to be modified. It was not possible to use the keyword "start state". Although we modified the parser/lexer look-ahead, it did not work, therefore we used "start_state". Furthermore we could not use double quotation marks, so we used single ones. In AntLR both situations worked without modification. This will of course have an impact on the syntactic noise.

Monticore generated collections with linear lookup time for states and transitions, therefore we added, as in the case of DSL2JDT, a hashtable cache.

Development Experience

The development was straight forward. Knowing AntLR one would not have problems with Monticore. Furthermore Monticore provided a rich documentation.

However there were some issues. The compiling process was done by the Monticore server. During our development the server crashed several times. It seems their software crashes frequently due to malformed client input. Every time we had to contact their administrator which resulted in various time lags.

Approximately four hours were used to the grammar file. Two hours were used to write the semantics.

Chapter 4

Benchmark Results

4.1 Results by Technology

4.1.1 Apache SCXML

General Data

- Valid output: true
- Syntactic Noise (Levenshtein distance): 770
- Java VM: OpenJDK 64-Bit Server VM
- Java version: $1.6.0_0$
- Benchmarking date: 29.09.09 12:41:29

Lines of Code

	total	generated	user written
original	168	0	168
effective	112	0	112

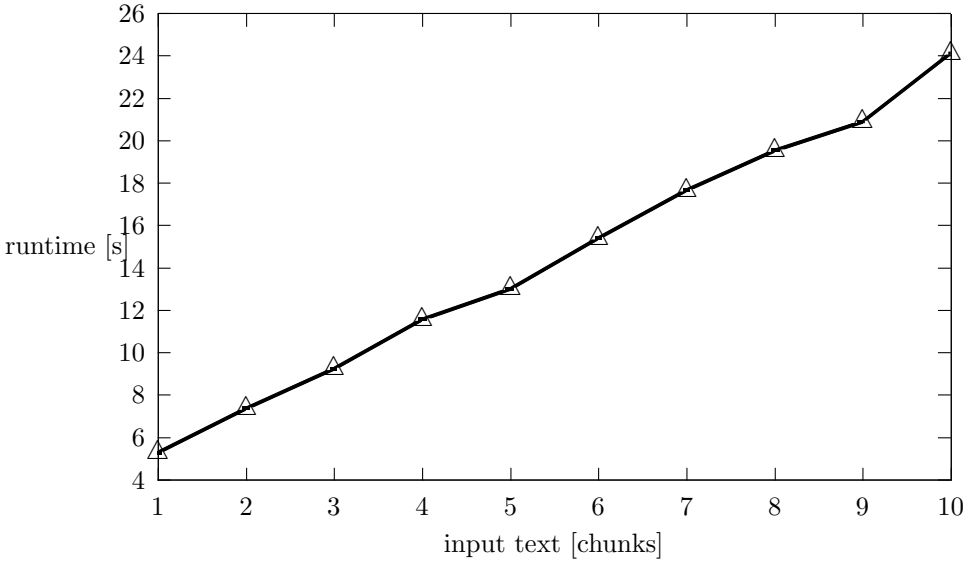
Characters of Code

	total	generated	user written
original	5462	0	5462
effective	4584	0	4584

Benchmark Runs

#	Text lenght [kb]	Runtime [ms]
1	51	5295
2	102	7388
3	153	9251
4	204	11561
5	255	13035
6	307	15405
7	358	17661
8	409	19550
9	460	20905
10	511	24115

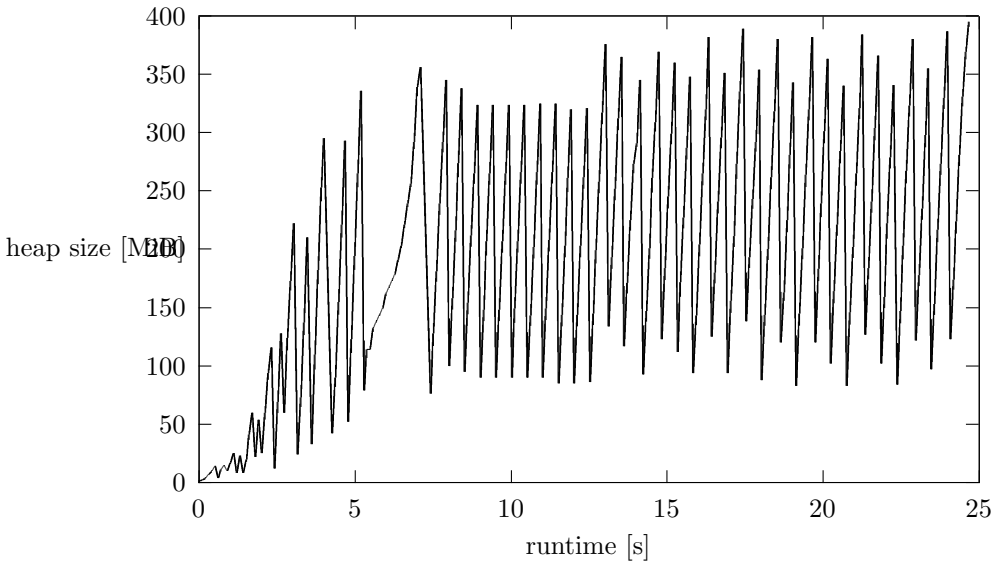
Runtime Diagram



Heap Allocation

Input Text [KiB]	512
Max Heap Size [MiB]	395
Garbage Collection Counts	46
Garbage Collection Runtime [ms]	766
Total Runtime [ms]	24791
$\frac{GarbageCollectionRuntime}{TotalRuntime}$ [ms]	0.031

Heap Allocation Diagram



4.1.2 Stratego/XT

4.1.3 AntLR

General Data

- Valid output: true
- Syntactic Noise (Levenshtein distance): 1
- Java VM: OpenJDK 64-Bit Server VM
- Java version: $1.6.0_0$
- Benchmarking date: 29.09.09 13:40:28

Lines of Code

	total	generated	user written
original	2068	1752	316
effective	1669	1443	226

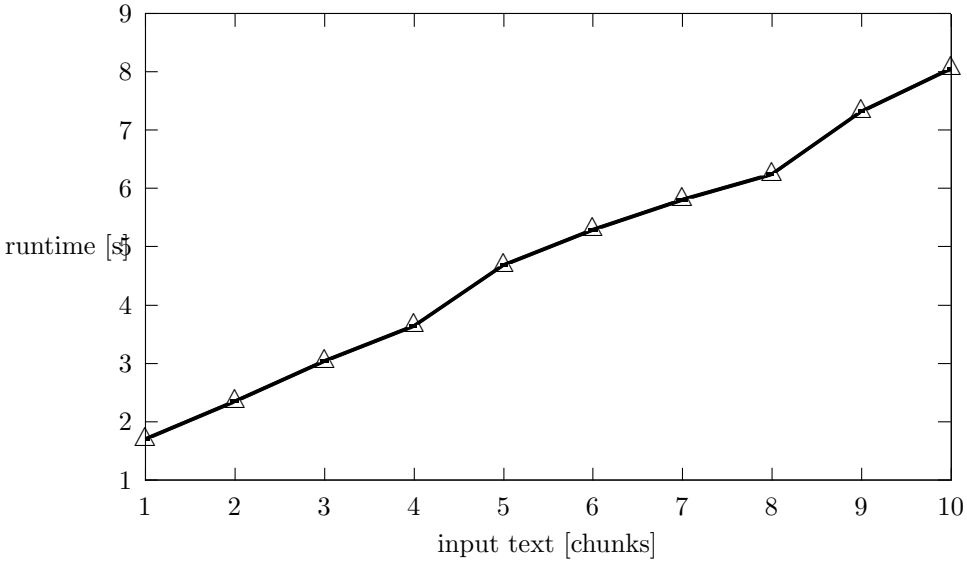
Characters of Code

	total	generated	user written
original	57271	51203	6068
effective	54567	48877	5690

Benchmark Runs

#	Text lenght [kb]	Runtime [ms]
1	51	1704
2	102	2353
3	153	3037
4	204	3650
5	255	4687
6	307	5292
7	358	5806
8	409	6241
9	460	7326
10	511	8051

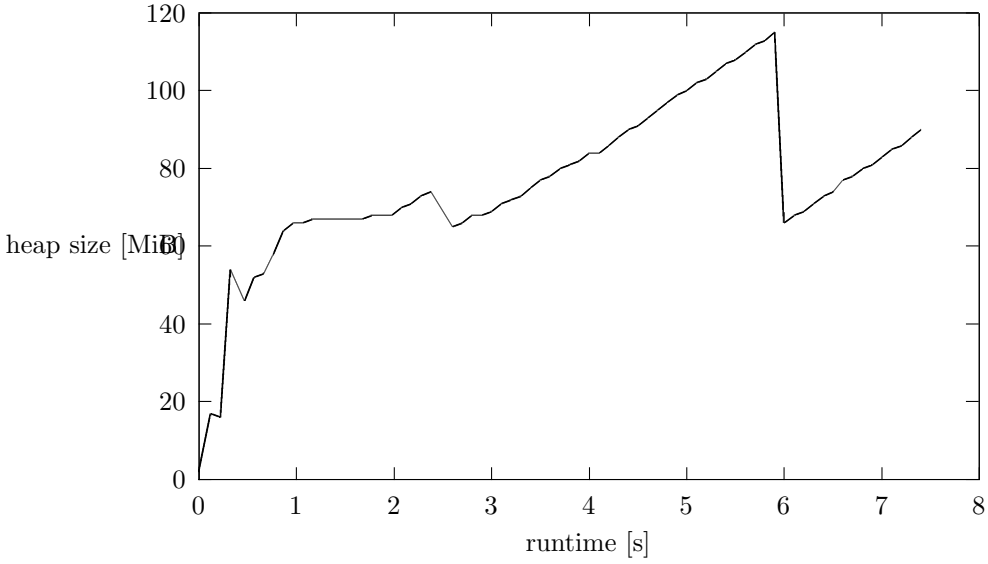
Runtime Diagram



Heap Allocation

Input Text [KiB]	512
Max Heap Size [MiB]	115
Garbage Collection Counts	5
Garbage Collection Runtime [ms]	143
Total Runtime [ms]	7510
$\frac{GarbageCollectionRuntime}{TotalRuntime}$ [ms]	0.019

Heap Allocation Diagram



4.1.4 Bison+Flex

General Data

- Valid output: true
- Syntactic Noise (Levenshtein distance): 0
- Java VM: OpenJDK 64-Bit Server VM
- Java version: $1.6.0_0$
- Benchmarking date: 29.09.09 12:46:26

Lines of Code

	total	generated	user written
original	4493	4011	482
effective	2904	2551	353

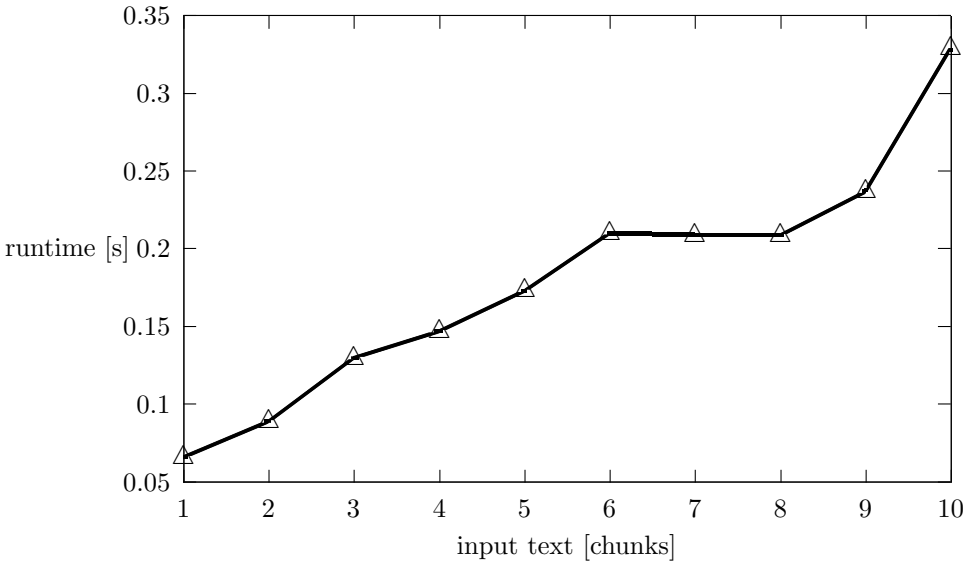
Characters of Code

	total	generated	user written
original	114655	104494	10161
effective	73573	65875	7698

Benchmark Runs

#	Text lenght [kb]	Runtime [ms]
1	51	66
2	102	89
3	153	130
4	204	147
5	255	173
6	307	210
7	358	209
8	409	209
9	460	237
10	511	329

Runtime Diagram



4.1.5 DSL2JDT

General Data

- Valid output: true
- Syntactic Noise (Levenshtein distance): 1060
- Java VM: OpenJDK 64-Bit Server VM
- Java version: $1.6.0_0$
- Benchmarking date: 29.09.09 12:46:44

Lines of Code

	total	generated	user written
original	3661	3405	256
effective	1416	1210	206

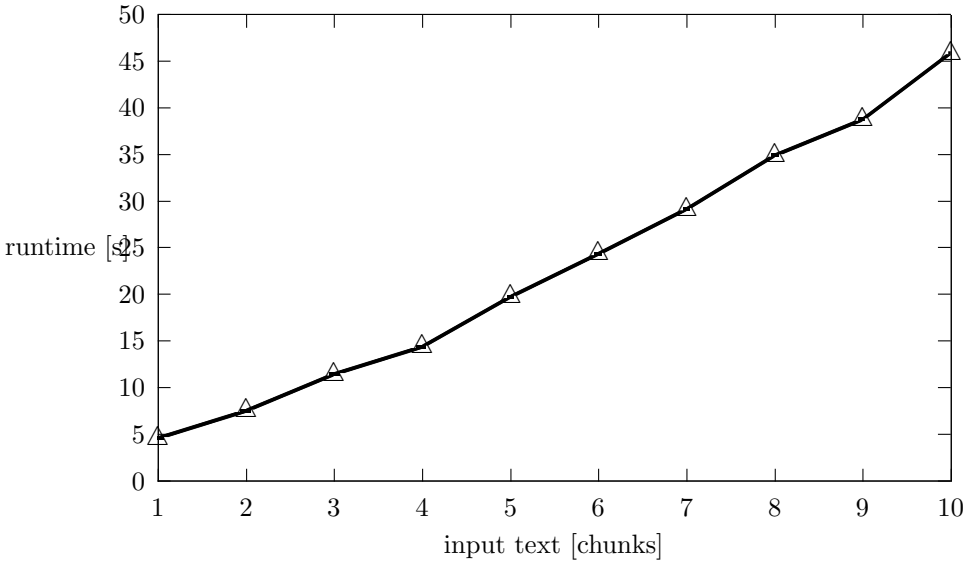
Characters of Code

	total	generated	user written
original	100750	93768	6982
effective	49723	43053	6670

Benchmark Runs

#	Text lenght [kb]	Runtime [ms]
1	51	4594
2	102	7522
3	153	11456
4	204	14436
5	255	19762
6	307	24337
7	358	29130
8	409	34929
9	460	38786
10	511	45904

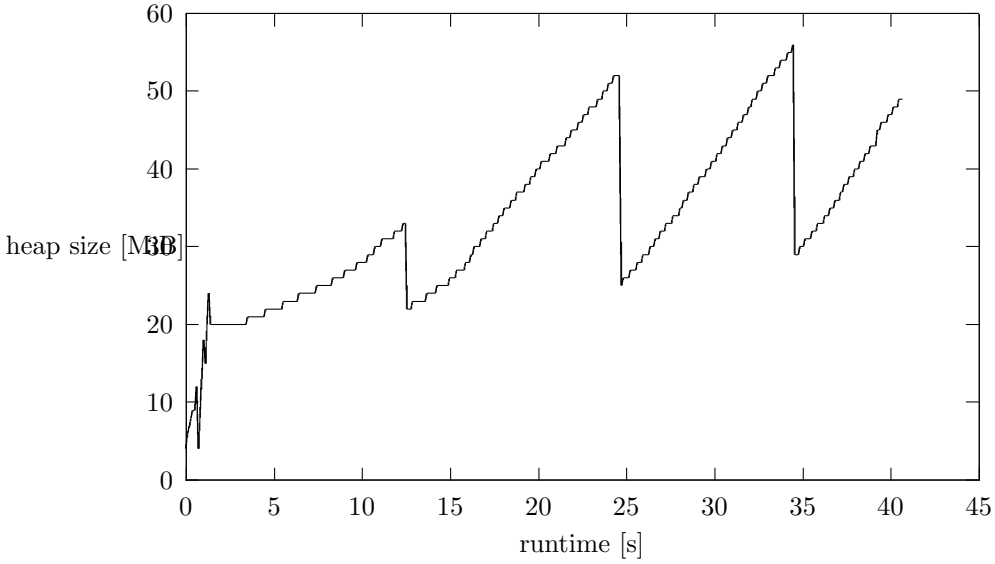
Runtime Diagram



Heap Allocation

Input Text [KiB]	512
Max Heap Size [MiB]	56
Garbage Collection Counts	6
Garbage Collection Runtime [ms]	82
Total Runtime [ms]	40772
$\frac{GarbageCollectionRuntime}{TotalRuntime}$ [ms]	0.0020

Heap Allocation Diagram



4.1.6 Monticore

General Data

- Valid output: true
- Syntactic Noise (Levenshtein distance): 14
- Java VM: OpenJDK 64-Bit Server VM
- Java version: $1.6.0_0$
- Benchmarking date: 29.09.09 12:52:25

Lines of Code

	total	generated	user written
original	5495	5243	252
effective	4006	3816	190

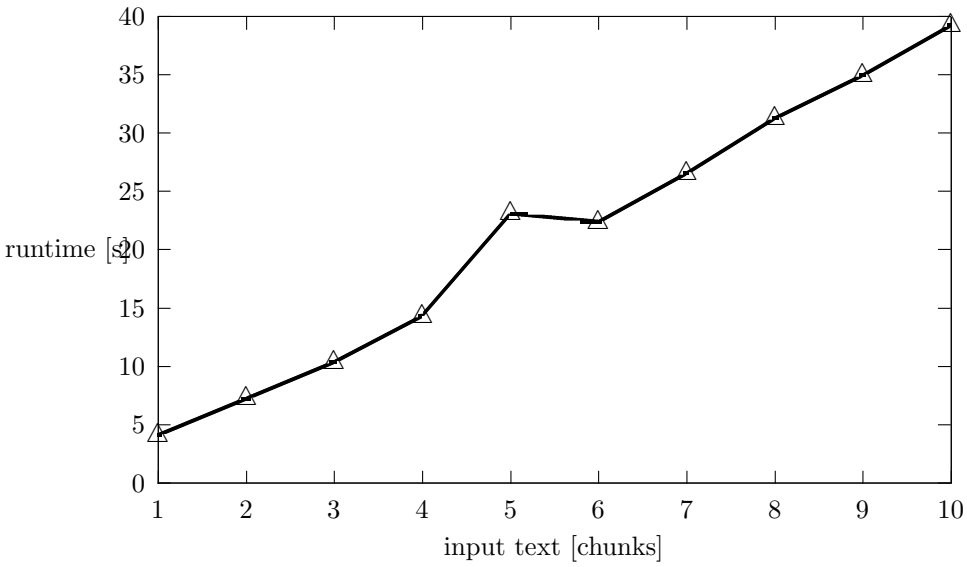
Characters of Code

	total	generated	user written
original	152955	147294	5661
effective	110508	105153	5355

Benchmark Runs

#	Text lenght [kb]	Runtime [ms]
1	51	4139
2	102	7268
3	153	10394
4	204	14329
5	255	23096
6	307	22373
7	358	26581
8	409	31285
9	460	34943
10	511	39234

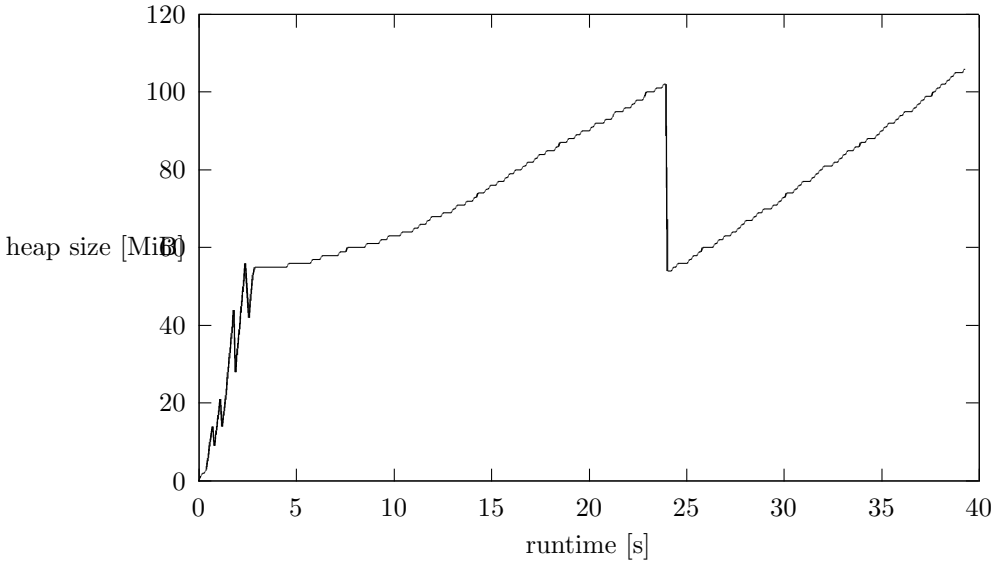
Runtime Diagram



Heap Allocation

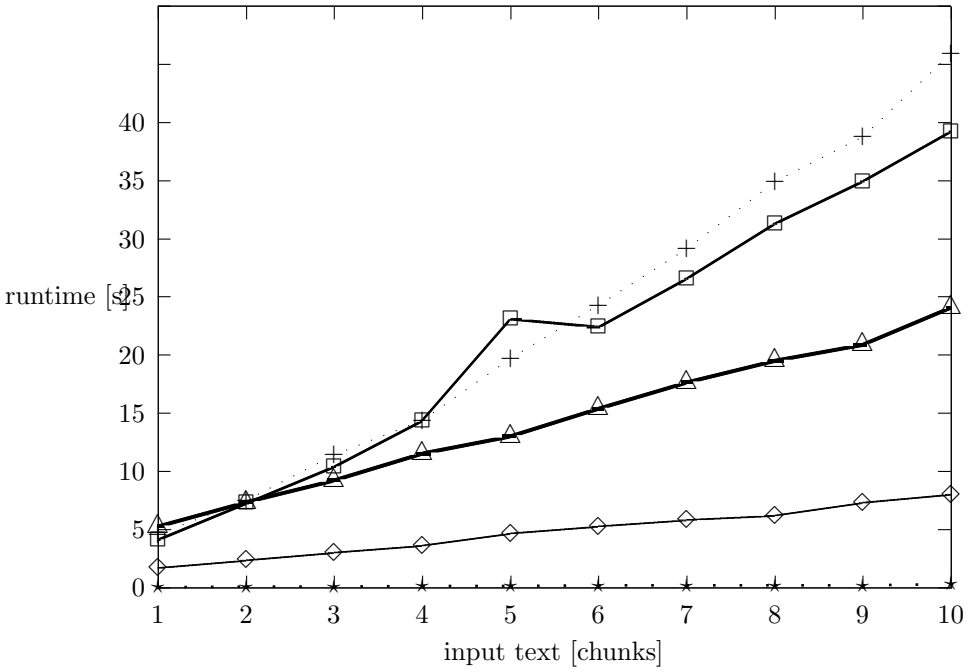
Input Text [KiB]	512
Max Heap Size [MiB]	106
Garbage Collection Counts	5
Garbage Collection Runtime [ms]	188
Total Runtime [ms]	39384
$\frac{GarbageCollectionRuntime}{TotalRuntime}$ [ms]	0.0048

Heap Allocation Diagram



4.2 Comparisons

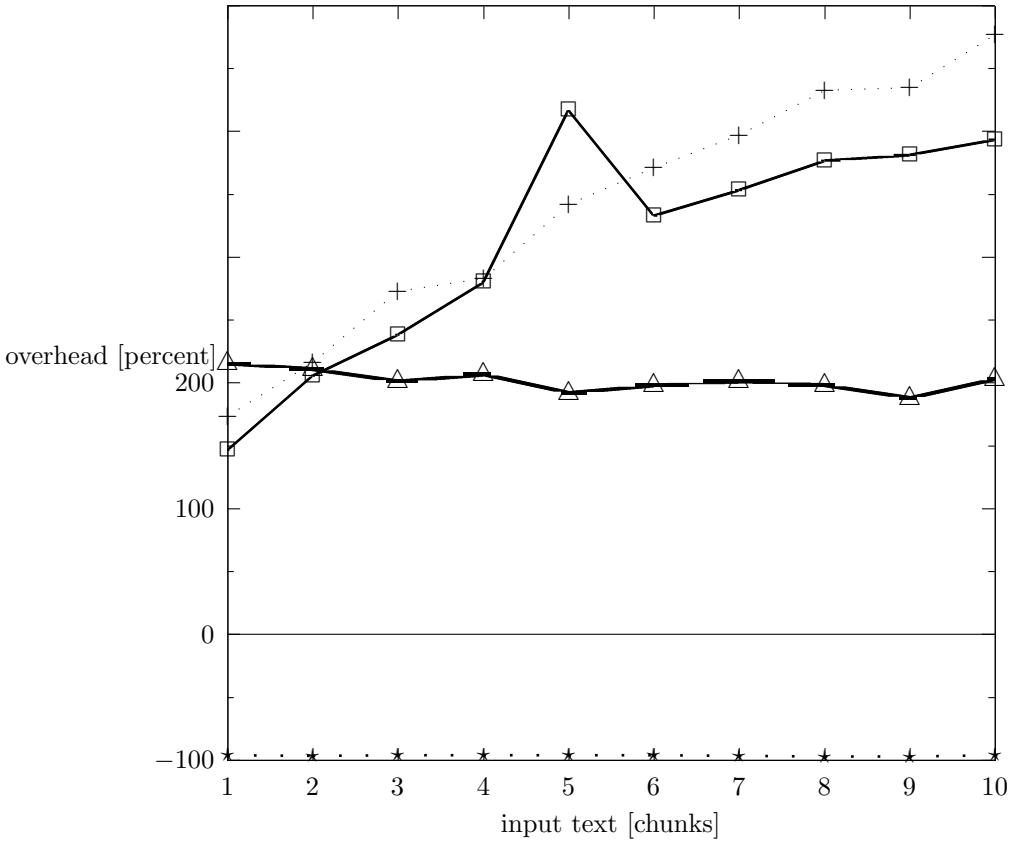
Runtime Comparison



Legend

AntLR	◇
DSL2JDT	+
Monticore	□
Bison+Flex	⋆
SCXML	△

Runtime Overhead relative to AntLR



AntLR	y=0
DSL2JDT	+
Monticore	□
Bison+Flex	⋆
SCXML	△

Bibliography

[1] Apache SCXML Website. http://commons.apache.org/scxml.

[2] Flex Website. http://flex.sourceforge.net.

[3] GNU Bison Website. http://www.gnu.org/software/bison.

[4] Monticore Website. http://www.monticore.de.

[5] Stratego XT. http://www.strategoxt.org.

[6] Miguel Garcia. Automating the embedding of Domain Specific Languages in Eclipse JDT. *Eclipse Corner Article*, 2008. Available at http://www.eclipse.org/articles/printable.php?file=Article-AutomatingDSLEmbeddings/index.html.

[7] Vladimir I. Levenshtein. *Binary codes capable of correcting deletions, insertions, and reversals.* Doklady Akademii Nauk SSSR, 1965.