A Semi-Automatically Assessable Design for a Hands-On Compilers Course

Omar Mohammad Othman, *German University in Cairo*

***Abstract*—This paper describes the author's experience with a hands-on compilers course designed and carried on in the German University in Cairo. The five main points are the course objectives, the course organization, the author's approach to formulating the lab exercises – an approach recommended for programming courses in general (which is also the main contribution as it offers a method for "semi-automatic" grading *without the need for an automatic assessment system*, besides documenting the author's experience categorizing and offering a hands-on compilers course), the small language "invented" for which the lexical analyzer, syntactic analyzer and interpreter are to be written by the students (which is the course's ongoing project) and the adopted grading scheme. Besides, having the course's project seamlessly integrated within the other exercises (as opposed to a separate bulk at the end of the course) is another point stressed in its design. An idea for a system for automating the whole process is proposed at the end.**

Keywords-Compilers; Semi-Automatic Assessment; Hands-On Course; ACM-ICPC Problems; Seen Language

I. Introduction

One of the fundamental problems in computer science education is adapting the exercises at hand to fit certain constraints in the everyday educational process – such as available equipment, available time and the need to grade the work of a large number of students in a limited amount of time. It is always challenging to design a course that fulfills the required scientific goals and is still easy for the instructor to maintain and grade. Grading is always a key factor regarding the success of the educational process. It is the only assessment method the instructor possesses to measure their success delivering the course content clearly and satisfactorily. Besides, it is often the case that some important questions are not posed in class simply because there is no clear vision for their grading. Clear grading schemes also assure academic fairness (ensuring students are treated equally), students' satisfaction and instructor unbiasedness.

The main motive behind writing this paper is that – to the author's knowledge – there is no publication out there in the literature that discusses the design of a hands-on compilers course with this point of easy and precise grading as the major design decision. Besides, documenting the course itself serves as a road map for other instructors teaching the same course.

II. Related Work

There are many systems out there for automatic assessment, surveyed in [1] and including – for example – those in [2] and [3]. The approach here – on the contrary – is not to use a system; it actually tries to avoid that burden (especially to new universities and/or those having no funds for creating their own or purchasing one). This is more similar to [4], where his system's automated part is "simulated" here by the use of the file comparison tool [6]. To the author's best knowledge, a paper that focuses solely on designing a hands-on compilers course is not available in today's literature.

III. Paper Organization

The course outlined in this paper was a hands-on lab course working in parallel with an undergraduate compilers course (thus serving as its practical part). The paper mainly discusses the course's objectives, methodology and organization – a concrete example of the author's vision regarding facilitating the administrative role of the instructor that has no access to an elaborate automatic assessment system, as opposed to the academic role (creating the syllabus itself). Meanwhile, the paper serves as a roadmap to a hands-on compilers course in particular (the material is available upon request to those interested in teaching the course in their universities… including the small language created to support achieving the goals of the course). The paper is organized as follows:

[Section 2 – Course Objectives] Lists the course's objectives which will guide its illustration in subsequent sections. Throughout the paper, goals are referred to by their names.

[Section 3 – Course Organization] The course's organization (the types of exercises encountered, their order and their categorization).

[Section 4 – Course Map] Illustrates how the course can actually be done in twelve weeks (once a week).

[Section 5 – Grading Schemes] Discusses the adopted grading process and suggests another, more practical one.

[Section 6 – Automation Option Proposal] Briefly describes an idea for automating the whole course management process.

The conclusion and the acknowledgements come at the end.

DOI: 10.5176/2251-3043_3.1.241

IV. Course Objectives

The objectives of this course are mainly five points, named here for easy reference throughout the paper:

|G1| Introducing an easy-to-follow track of exercises.

|G2| Making exercises (and even the project) clear and strict regarding grading scheme.

|G3| Creating an interesting environment for attracting the students to the relatively hard compilers hands-on course.

|G4| Avoiding having a burden at the end of the course when most students are busy with the projects of other courses.

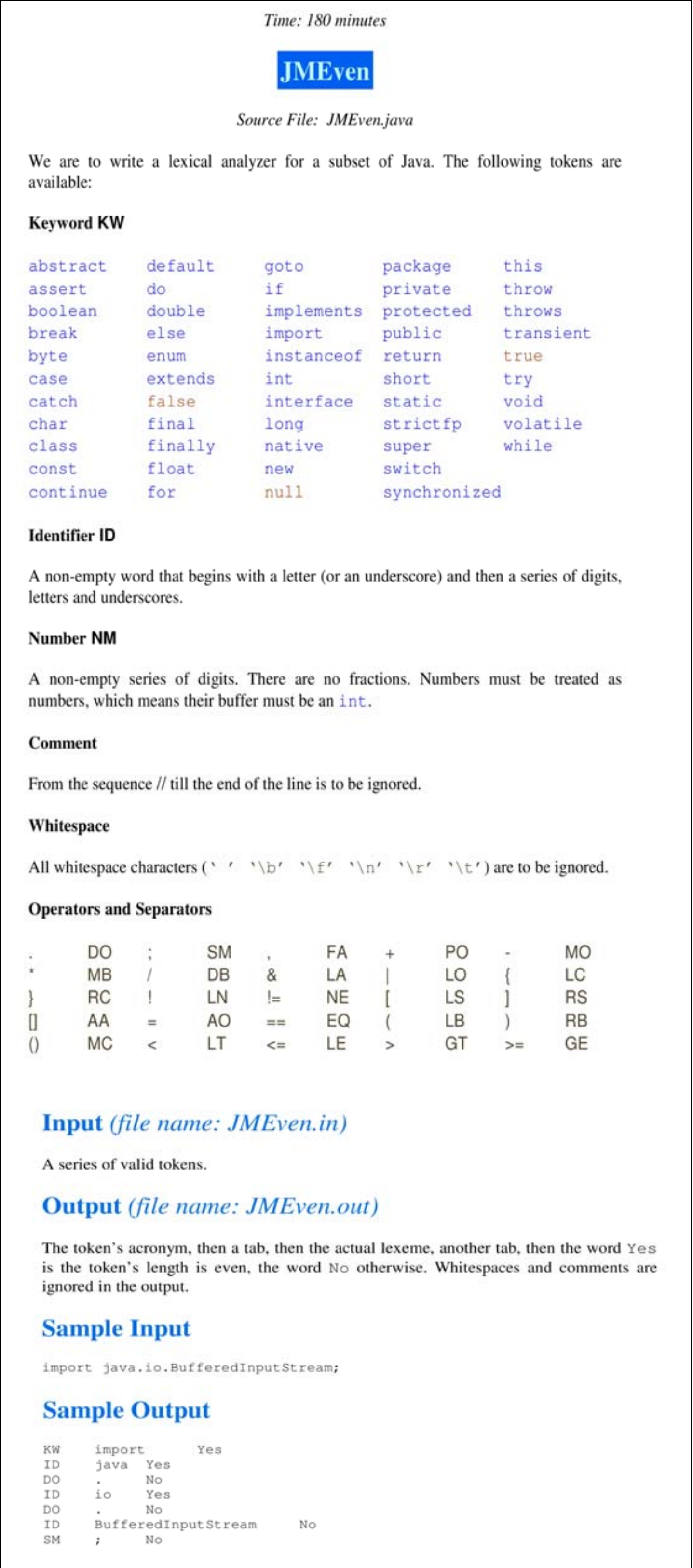
|G5| Constraining the work to be on campus (even the project) so as to improve the skills of working under pressure and meeting strict deadlines, besides organizing students' time.

A. *Adopting ACM-ICPC Problem Style*

To achieve G1 and G2, the ACM-ICPC [5] problem style was adopted. Each problem has a very clear set of rules, a fixed-format input and output (usually via text files) judged automatically – simply by comparing the correct file vs. the file output from running the submitted code. In ACM-ICPC [5] the correct files (both input and output) are always unseen, but to avoid making things quite hard; students always had the correct files (both input and output) so as to continuously know the "cases" their codes fall in. The files can be easily compared using a file comparison tool like *KDiff3* [6], available on their machines. Mimicking the ACM-ICPC [5] competition style in the lab also helps satisfying G3, where there is a continuous sense of competition every week to finish first and score a certain bonus. Besides, students know their task from Day 1: The exercise is to be solved by producing a correct output file and that's all. Another plus adopting this approach is to develop the "software engineering sense" of meeting requirements exactly – in this context a single space or period (or even a case change) is simply a sort of "wrong answer" and the code has to be modified (more details on grading are discussed in Section 5: Grading Scheme). Yet another advantage is the varying sophistication of the same exercise (from easy to very hard) just by varying the input. For example, in lexical analysis exercises; one can list in the input file all possible combinations of tokens (generated automatically using two nested loops). This is a very hard exercise where any minor error in the code will be undoubtedly detected, and it actually happened in an exercise on lexical analysis that one of the students had a single wrong case out of more than 25000 cases (here tokens) in the input file! Obviously this exercise can be set easier by avoiding listing tricky combinations of tokens in the input file.

B. *Handling Time Constraints*

To achieve G5, a "lab manual" is always sent to the students 5 – 7 days before the session. Hence any theoretical part is to be revised beforehand, thus the work is always supposed to be started and finished in the lab. The exercise itself is always unseen.



Time: 180 minutes

JMEven

Source File: JMEven.java

We are to write a lexical analyzer for a subset of Java. The following tokens are available:

Keyword KW

abstract	default	goto	package	this
assert	do	if	private	throw
boolean	double	implements	protected	throws
break	else	import	public	transient
byte	enum	instanceof	return	true
case	extends	int	short	try
catch	false	interface	static	void
char	final	long	strictfp	volatile
class	finally	native	super	while
const	float	new	switch	
continue	for	null	synchronized	

Identifier ID

A non-empty word that begins with a letter (or an underscore) and then a series of digits, letters and underscores.

Number NM

A non-empty series of digits. There are no fractions. Numbers must be treated as numbers, which means their buffer must be an `int`.

Comment

From the sequence // till the end of the line is to be ignored.

Whitespace

All whitespace characters (`' '` `'\b'` `'\f'` `'\n'` `'\r'` `'\t'`) are to be ignored.

Operators and Separators

.	DO	;	SM	,	FA	+	PO	-	MO
*	MB	/	DB	&	LA	|	LO	{	LC
}	RC	!	LN	!=	NE	[	LS	]	RS
[]	AA	=	AO	==	EQ	(	LB	)	RB
()	MC	<	LT	<=	LE	>	GT	>=	GE

Input *(file name: JMEven.in)*

A series of valid tokens.

Output *(file name: JMEven.out)*

The token's acronym, then a tab, then the actual lexeme, another tab, then the word `Yes` is the token's length is even, the word `No` otherwise. Whitespaces and comments are ignored in the output.

Sample Input

```
import java.io.BufferedInputStream;
```

Sample Output

```
KW    import        Yes
ID    java  Yes
DO    .     No
ID    io    Yes
DO    .     No
ID    BufferedInputStream       No
SM    ;     No
```

Figure 1. Typical J-Series, M-Flavor Exercise

V. Course Organization

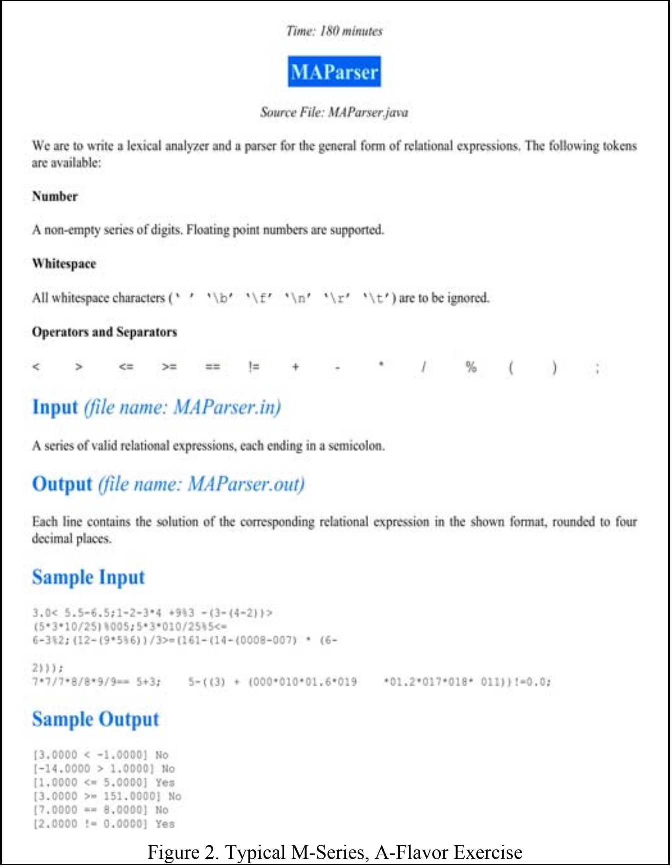
To achieve G1 together with G4, the entire course is based on three "series" of exercises: The **J-Series**, the **M-Series** and the **X-Series**, each subdivided into the **M-Flavor** (everything is written from scratch – **M** for Manual) and the **A-Flavor** (a generator is used: *JLex* [7] for generating lexical analyzers and *CUP* [8] for generating parsers, **A** for Automatic). Thus the same exercise always has two flavors, for achieving both goals… thorough concepts grasping and practicality.

A. *The J-Series*

These are exercises that use **J**ava (which is the adopted language in GUC), and thus is suitable for massy exercises involving a large number of varying tokens (mainly lexical analysis) – besides being known to the students. *Figure 1* is an example of an exercise from the **J-Series**, **M-Flavor**.

B. *The M-Series*

The **M**ath series is the most commonly used for compiler exercises due to its simplicity, as arithmetic formulae possess a simple lexical structure yet demonstrate fundamental issues regarding parsing – mainly those of operator precedence and associativity – besides balancing structures (brackets). *Figure 2* is a complete example of an exercise from the **M-Series**, **A-Flavor**. *Figure 3* is a more advanced problem from the **M-Series**, **M-Flavor** that focuses on building the abstract syntax tree before evaluation.



Time: 180 minutes

MAParser

Source File: MAParser.java

We are to write a lexical analyzer and a parser for the general form of relational expressions. The following tokens are available:

Number

A non-empty series of digits. Floating point numbers are supported.

Whitespace

All whitespace characters (`' '` `'\b'` `'\f'` `'\n'` `'\r'` `'\t'`) are to be ignored.

Operators and Separators

`<` `>` `<=` `>=` `==` `!=` `+` `-` `*` `/` `%` `(` `)` `;`

Input *(file name: MAParser.in)*

A series of valid relational expressions, each ending in a semicolon.

Output *(file name: MAParser.out)*

Each line contains the solution of the corresponding relational expression in the shown format, rounded to four decimal places.

Sample Input

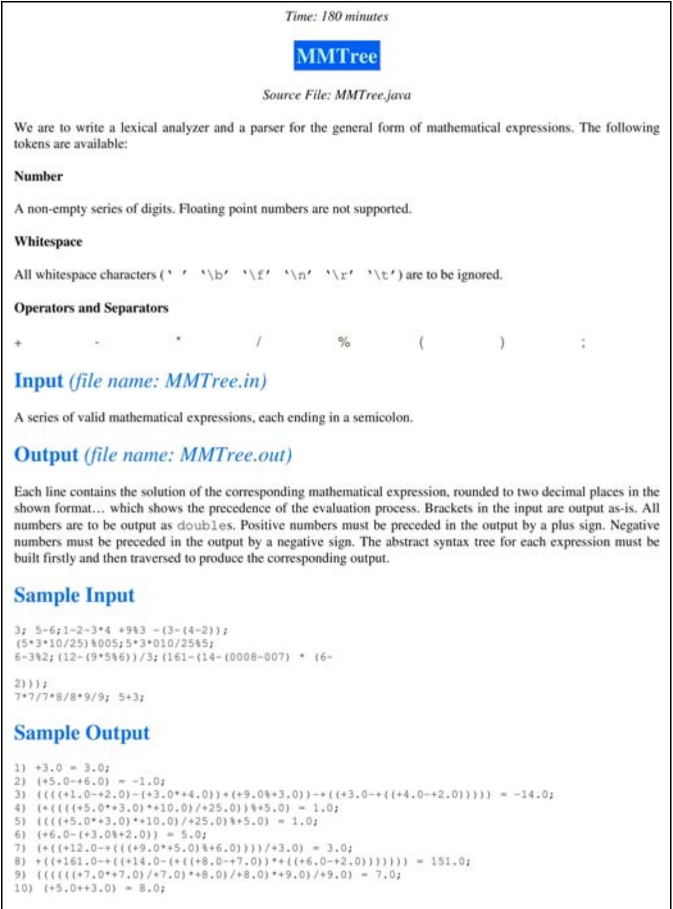
```
3.0< 5.5-6.5;1-2-3*4 +9%3 -(3-(4-2))>
(5*3*10/25)%005;5*3*010/25%5<=
6-3%2;(12-(9*5%6))/3>=(161-(14-(0008-007) * (6-

2)));
7*7/7*8/8*9/9== 5+3;    5-((3) + (000*010*01.6*019    *01.2*017*018* 011))!=0.0;
```

Sample Output

```
[3.0000 < -1.0000] No
[-14.0000 > 1.0000] No
[1.0000 <= 5.0000] Yes
[3.0000 >= 151.0000] No
[7.0000 == 8.0000] No
[2.0000 != 0.0000] Yes
```

Figure 2. Typical M-Series, A-Flavor Exercise



Time: 180 minutes

MMTree

Source File: MMTree.java

We are to write a lexical analyzer and a parser for the general form of mathematical expressions. The following tokens are available:

Number

A non-empty series of digits. Floating point numbers are not supported.

Whitespace

All whitespace characters (`' '` `'\b'` `'\f'` `'\n'` `'\r'` `'\t'`) are to be ignored.

Operators and Separators

`+` `-` `*` `/` `%` `(` `)` `;`

Input *(file name: MMTree.in)*

A series of valid mathematical expressions, each ending in a semicolon.

Output *(file name: MMTree.out)*

Each line contains the solution of the corresponding mathematical expression, rounded to two decimal places in the shown format… which shows the precedence of the evaluation process. Brackets in the input are output as-is. All numbers are to be output as `doubles`. Positive numbers must be preceded in the output by a plus sign. Negative numbers must be preceded in the output by a negative sign. The abstract syntax tree for each expression must be built firstly and then traversed to produce the corresponding output.

Sample Input

```
3; 5-6;1-2-3*4 +9%3 -(3-(4-2));
(5*3*10/25)%005;5*3*010/25%5;
6-3%2;(12-(9*5%6))/3;(161-(14-(0008-007) * (6-

2)));
7*7/7*8/8*9/9; 5+3;
```

Sample Output

```
1) +3.0 = 3.0;
2) (+5.0-+6.0) = -1.0;
3) ((((+1.0-+2.0)-(+3.0*+4.0))+(+9.0%+3.0))-+((+3.0-+((+4.0-+2.0))))) = -14.0;
4) (+((((+5.0*+3.0)*+10.0)/+25.0))%+5.0) = 1.0;
5) ((((+5.0*+3.0)*+10.0)/+25.0)%+5.0) = 1.0;
6) (+6.0-(+3.0%+2.0)) = 5.0;
7) (+((+12.0-+(((+9.0*+5.0)%+6.0))))/+3.0) = 3.0;
8) +((+161.0-+((+14.0-(+((+8.0-+7.0))*+((+6.0-+2.0))))))) = 151.0;
9) ((((((+7.0*+7.0)/+7.0)*+8.0)/+8.0)*+9.0)/+9.0) = 7.0;
10) (+5.0++3.0) = 8.0;
```

Figure 3. Advanced M-Series, M-Flavor Exercise

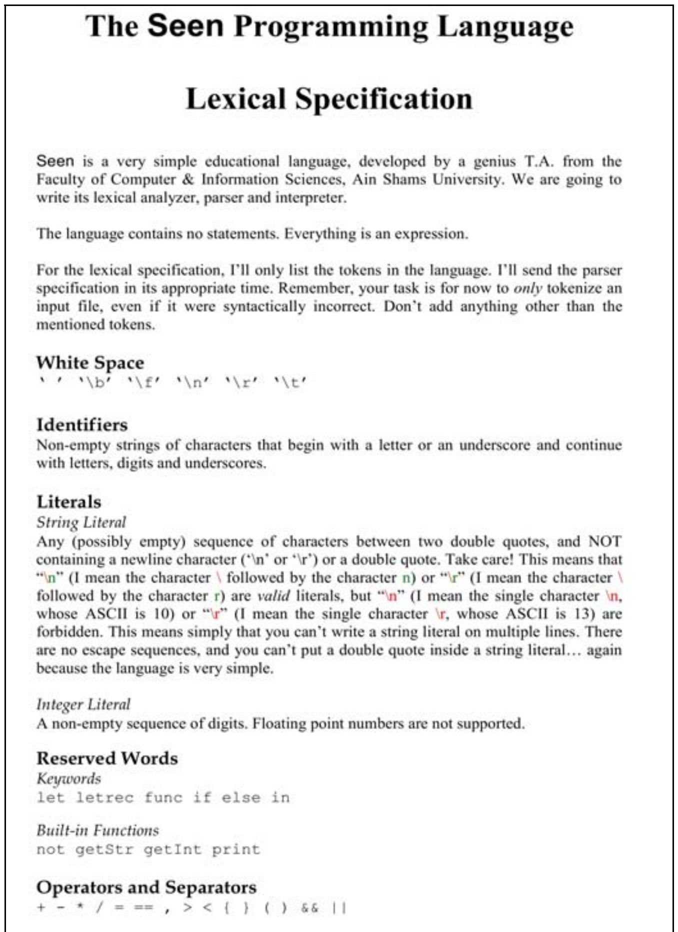
C. *The X-Series*

This is the series where students implement their project. Three phases are required: The lexical analyzer, the parser and the interpreter of a small language called Seen, which was specifically created for being used in such courses (Seen is how the Arabic letter س is pronounced, which is used for unknowns like X in English, and hence the name of the series).

Seen is somewhat similar to Lisp [9], in the sense that it is dynamically-typed, supports lambda-expressions and closures, and is expression-based (as opposed to statement-based). The Seen language is more or less Scheme [13] with a more "natural" syntax as opposed to the bracket-based Scheme [13] style. It was chosen among other Lisp [9] dialects due to the support of lexical scoping (evaluating the function expression in a new environment extending the definition environment). The Scheme [13] language specification is available at [14]. In brief, the language basically contains:

[Literal] A literal is an expression that evaluates to itself. One can only use integer literals and string literals in Seen.

[Operation] Arithmetic operations (addition, subtraction, multiplication and division), relational operations (less than, greater than and the equality test), and logical operations (conditional OR and conditional AND).



The Seen Programming Language

Lexical Specification

Seen is a very simple educational language, developed by a genius T.A. from the Faculty of Computer & Information Sciences, Ain Shams University. We are going to write its lexical analyzer, parser and interpreter.

The language contains no statements. Everything is an expression.

For the lexical specification, I'll only list the tokens in the language. I'll send the parser specification in its appropriate time. Remember, your task is for now to *only* tokenize an input file, even if it were syntactically incorrect. Don't add anything other than the mentioned tokens.

White Space

```
' ' '\b' '\f' '\n' '\r' '\t'
```

Identifiers

Non-empty strings of characters that begin with a letter or an underscore and continue with letters, digits and underscores.

Literals

String Literal

Any (possibly empty) sequence of characters between two double quotes, and NOT containing a newline character ('\n' or '\r') or a double quote. Take care! This means that "\n" (I mean the character \ followed by the character n) or "\r" (I mean the character \ followed by the character r) are *valid* literals, but "\n" (I mean the single character \n, whose ASCII is 10) or "\r" (I mean the single character \r, whose ASCII is 13) are forbidden. This means simply that you can't write a string literal on multiple lines. There are no escape sequences, and you can't put a double quote inside a string literal… again because the language is very simple.

Integer Literal

A non-empty sequence of digits. Floating point numbers are not supported.

Reserved Words

Keywords

```
let letrec func if else in
```

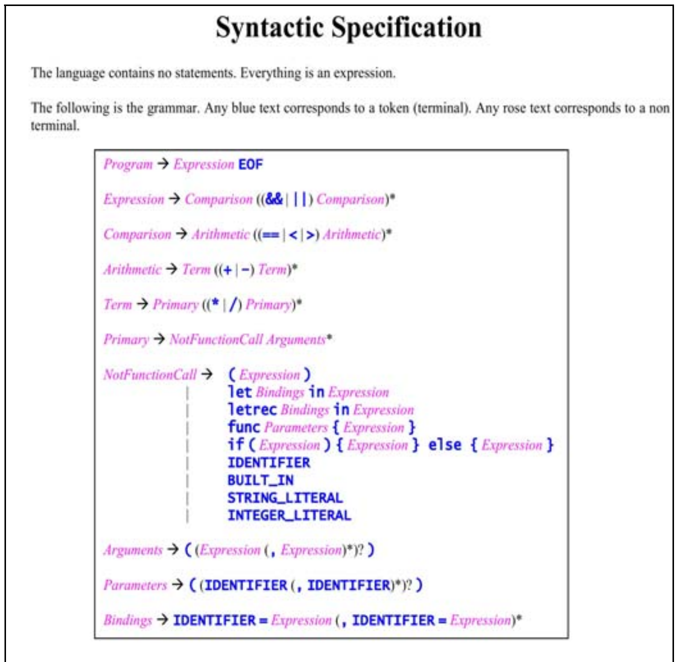
Built-in Functions

```
not getStr getInt print
```

Operators and Separators

```
+ - * / = == , > < { } ( ) && ||
```

Figure 4. Seen's Lexical Specification



Syntactic Specification

The language contains no statements. Everything is an expression.

The following is the grammar. Any blue text corresponds to a token (terminal). Any rose text corresponds to a non terminal.

```
Program → Expression EOF

Expression → Comparison ((&& | ||) Comparison)*

Comparison → Arithmetic ((== | < | >) Arithmetic)*

Arithmetic → Term ((+ | -) Term)*

Term → Primary ((* | /) Primary)*

Primary → NotFunctionCall Arguments*

NotFunctionCall →   ( Expression )
                |   let Bindings in Expression
                |   letrec Bindings in Expression
                |   func Parameters { Expression }
                |   if ( Expression ) { Expression } else { Expression }
                |   IDENTIFIER
                |   BUILT_IN
                |   STRING_LITERAL
                |   INTEGER_LITERAL

Arguments → ( (Expression (, Expression)*)? )

Parameters → ( (IDENTIFIER (, IDENTIFIER)*)? )

Bindings → IDENTIFIER = Expression (, IDENTIFIER = Expression)*
```

Figure 5. Seen's Syntactic Specification

[If Expression] The condition is evaluated. A zero result gives the expression the value of the else part. Otherwise the expression is assigned the value of the if part (this is the behavior of the conditional operator ? : in C). Note that – unlike conventional languages – this is an expression, not a statement; which means its value can be assigned to a variable.

[Lambda Expression] The same as lambda expressions in Scheme [13], but the keyword is called func. This expression evaluates to a function object.

[Function Application] Invoking the function object created by the *Lambda Expression*. Actual parameters are evaluated, formal parameters are substituted by calculated arguments and the expression is then evaluated as a whole.

[Let Expression] The language does not support assignment operations. A variable is given a value through "binding" it in the let part and using it in the in part.

[Letrec Expression] A flavor of the *Let Expression* that supports recursion.

1) *Lexical Analyzer*

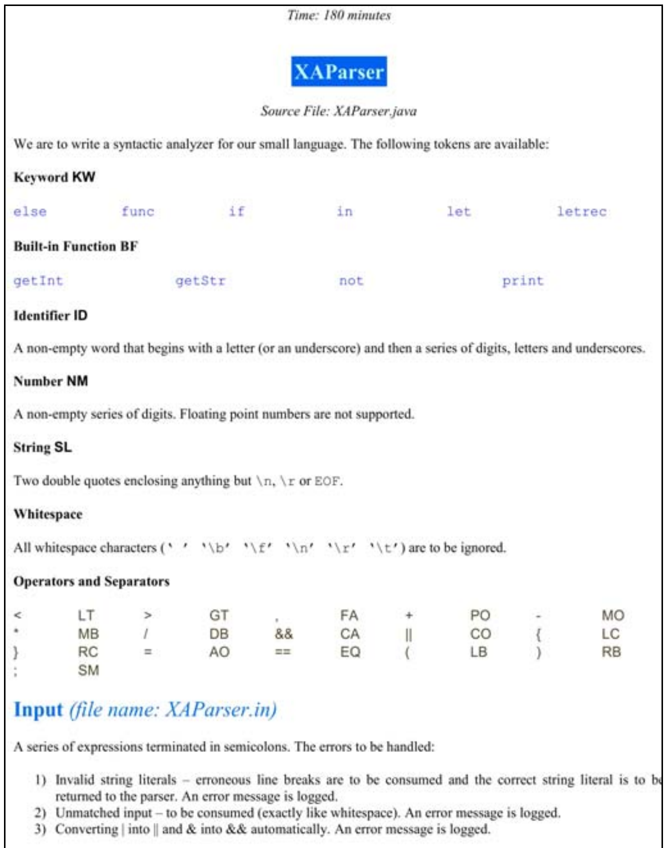
The lexical analyzer is done after two labs on lexical analysis (one written from scratch and one using *JLex* [7]). *Figure 4* is the lexical specification for Seen. One of the important exercises was to design a lexical analyzer that is capable of detecting lexical errors and actually correcting them. Three types of errors were handled: (i) A strange character (ignored + informative message), (ii) | instead of || for representing the OR operator or & instead of && for representing the AND operator (completed and returned normally to the parser + informative message) and (iii) erroneous string literals having line breaks inside (line breaks removed automatically and the string literal returned normally to the parser + informative message).

2) *Syntactic Analyzer*

The syntactic analyzer (parser) is done after two labs on parsing (one written from scratch and one using *CUP* [8]). *Figure 5* is the syntactic specification (context-free grammar) for Seen. One of the important exercises was to design a parser that is capable of detecting parse errors and actually correcting them. For error recovery, the "panic mode" technique discussed in the famous *Dragon Book* [10] was adopted. The language was extended a little bit to be a series of expressions terminated by a semicolon rather than a single expression. Upon detecting an error, the rest of the tokens till the next "synchronizing token" (the semicolon in our case) were skipped, an error message issued, and parsing was continued starting from the next statement. If the error were a missing terminal, it was automatically inserted, an informative message issued and parsing the same statement continued normally. This is usually not the case with real compilers (which normally refrain from changing anything), but the exercise gives true insight into how much powerful the technique is.

3) *Interpreter*

The interpreter is done on the last day of the course. The "lab manual" is a collection of three video lectures given in a previous course in the Faculty of Computer and Information Sciences, Ain Shams University as an informal introduction to programming languages and interpreters – during which the Seen language and the complete steps for writing the lexical analyzer, syntactic analyzer and interpreter were illustrated (the video lectures are available upon request. They are in Arabic, however). It is noteworthy that doing the project labs in the aforementioned times helped achieve G4, because at the end of the semester, only one last mainstream lab was needed to finalize the project.



Time: 180 minutes

XAParser

Source File: XAParser.java

We are to write a syntactic analyzer for our small language. The following tokens are available:

Keyword KW

`else` `func` `if` `in` `let` `letrec`

Built-in Function BF

`getInt` `getStr` `not` `print`

Identifier ID

A non-empty word that begins with a letter (or an underscore) and then a series of digits, letters and underscores.

Number NM

A non-empty series of digits. Floating point numbers are not supported.

String SL

Two double quotes enclosing anything but `\n`, `\r` or `EOF`.

Whitespace

All whitespace characters (`' '` `'\b'` `'\f'` `'\n'` `'\r'` `'\t'`) are to be ignored.

Operators and Separators

<	LT	>	GT	,	FA	+	PO	-	MO
*	MB	/	DB	&&	CA	||	CO	{	LC
}	RC	=	AO	==	EQ	(	LB	)	RB
;	SM								

Input *(file name: XAParser.in)*

A series of expressions terminated in semicolons. The errors to be handled:

1) Invalid string literals – erroneous line breaks are to be consumed and the correct string literal is to b
returned to the parser. An error message is logged.
2) Unmatched input – to be consumed (exactly like whitespace). An error message is logged.
3) Converting | into || and & into && automatically. An error message is logged.

Figure 6. Seen's Error Recovery

VI. COURSE MAP

After highlighting each of the three series, the course map can be followed easily:

[Lab 0] Simplified ACM-ICPC-Style Programming Contest – (Exercise: Miscellaneous). This is just a programming reminder and an example of how future labs are organized.

[Lab LM] "Lexical – Manual" A handwritten lexical analyzer (Exercise: J-Series, M-Flavor).

[Lab LA] "Lexical – Automatic" A lexical analyzer using *JLex* [7] (Exercise: J-Series, A-Flavor).

[Lab PLM] "Project – Lexical – Manual" A handwritten lexical analyzer for Seen (Exercise: X-Series, M-Flavor).

[Lab PLA] "Project – Lexical – Automatic" A lexical analyzer for Seen using *JLex* [7] (Exercise: X-Series, A-Flavor).

[Lab SM] "Syntactic – Manual" A handwritten syntactic analyzer <<predictive parser>> (Exercise: M-Series, M-Flavor).

[Lab SA] "Syntactic – Automatic" A syntactic analyzer using *CUP* [8] (Exercise: M-Series, A-Flavor).

[Lab PSM] "Project – Syntactic – Manual" A handwritten syntactic analyzer for Seen (Exercise: X-Series, M-Flavor. See *Figure 6*. The sample input and output are not shown).

[Lab PSA] "Project – Syntactic – Automatic" A syntactic analyzer for Seen using *CUP* [8] (Exercise: X-Series, A-Flavor). This was the hardest exercise.

[Lab ST] "Syntactic – Tree" A handwritten syntactic analyzer (Exercise: M-Series, M-Flavor). The difference is that the abstract syntax tree must be built explicitly and then traversed to produce the correct output. This is as opposed to solving during predictive parsing which is essentially building a "logical" abstract syntax tree. This lab is important because all subsequent compilation phases (semantic analysis, intermediate code generation, code optimization and final code generation) depend on this data structure, thus at least one lab must tackle this point.

[Lab PIM] "Project – Interpreter – Manual" A handwritten interpreter for Seen (Exercise: X-Series, M-Flavor).

[Lab PIA] "Project – Interpreter – Automatic" An interpreter for Seen using *CUP* [8] (Exercise: X-Series, A-Flavor). This one is optional (just for completeness). We didn't do it.

VII. GRADING SCHEMES

For grading the lab exercises, the hard method of completing the missing code (or debugging the code if only some of the cases in the final output file failed) was adopted. The student's grade was reduced for every added "piece of code" for solving a certain problem. Comments were inserted in accordance with the added code so that the student always knew his mistakes, thus both understanding the rationale behind the grade and achieving the utmost scientific benefit as the code was always massive (and thus knowing all the mistakes was always enlightening).

A small point here to note is that this grading style was always "psychologically" outstanding regarding those students who completed most of the work but had some bugs. It happened more than once that the students were delighted to know their bugs, especially those very tricky ones like the aforementioned student who had a single incorrect case out of more than 25000! Obviously, an alternative less time consuming method is just to estimate the missing code and reduce the grade accordingly, as opposed to actually completing the code.

Another suitable grading scheme may be to adopt a certain grade (e.g. 70%) for a reasonable output file, and then dividing the remaining 30% equally among the cases. So – for example – an input file with 300 cases has a grade of 0.1% per case, and a student with 280 correct cases acquires 98% (70% + 28%). Note that KDiff3 [6] is used to visualize the wrong cases, hence even this part can be easily automated. It's important to note that this is rarely practical. In most cases, either everything (but for some silly mistakes) is output, or nothing at all.

Another option (which is more appropriate as the cases will always have different levels of difficulty) is to weight the cases according to their hardness. That code that does not produce an output file at all can be graded using the previous approach: either completing the code or estimating the missing parts and reducing the grade according to the actions taken.

VIII. AUTOMATION OPTION PROPOSAL

Three main "phases" are often present in most hands-on courses. These are exercises' submission, plagiarism detection and grading. Automating the submission phase is easy – a web application can be implemented that organizes the whole process in a completely automated way. Plagiarism detection also has its tools (e.g. The Plagiarism Checker [11] and Plagiarismdetect [12]) and the research in this field is still active. Automatic grading is a much more involved option. It may be interesting to investigate the option of implementing an intelligent system programmed with "common errors" so that a portion of incorrect submissions becomes also automatically handled. Such a system may be infeasible if it is to be implemented as a general-purpose system. A better option is to implement a "course-specific" system that can only grade – for example – the twelve problems offered in this paper's course. Adopting the ACM-ICPC-Style of problems partially supports this point because of the fixed format of the output, and thus the clear goal of the program code.

IX. Conclusion

The main objective is to encourage instructors involved in programming courses to adopt the proposed style. The most important point is to formulate the exercise at hand into an ACM-ICPC-Style problem. Doing that is supposed to result in an easy-to-manage course (for the instructor) that is real benefit and also fun (for the students). Manageability comes primarily from the ease and accuracy of grading this style of exercises without a real need for an automatic assessment system, besides having a uniform course easily tracked by the students. Regarding compilers courses in particular, instructors are encouraged to divide the work constantly between handwritten and automatically generated modules in an interleaved fashion so that the students understand the concepts well while having a tool at hand to facilitate accomplishing the task and to actually use in real life. It's also very important to modularize the course's project and do it incrementally throughout the whole course so as to have the utmost understanding of each phase (as opposed to having the whole project implemented at the end). Another interesting point is the use of an expression-based language like Seen, mainly because of conciseness and clarity – besides its attractiveness for the students to have at the end of the course a complete "compiler" that accepts user input and shows results, even if the user supplies wrong inputs (if the error recovery module is to be implemented). This makes the student in front of a "real" product that he made himself, giving him a sort of self-satisfaction and confidence (some students actually made this comment).

Acknowledgments

I would like to deeply thank my genius friend, Mohamed Samy – Seen's inventor – who gave me the exclusive rights to use, modify and reimplement Seen in Java, inspired me to create this course and reviewed this paper thoroughly and patiently. I would like to thank Prof. Dr. Carmen Gervet (the course's supervisor) for giving me the complete freedom to design the course according to my vision, and Prof. Dr. Slim Abdennadher (the faculty's study dean) for patiently supporting me to change the course's initial official plans and for his efforts to solve the administrative problems that aroused during the semester. I would really like to thank all my students who were the main reason this course was a real success (and even fun). They exerted great efforts in each and every lab, and insisted on finishing all exercises even after staying for several hours in the lab (even after the allowed lab duration stated in the exercise is over). Special thanks go to the perfect student, Amr Labib for his extraordinary performance and to Heba Hesham for her extreme persistence and amazing hard work.

References

[1] C. Douce, D. Livingstone, J. Orwell. Article 4. *Journal on Educational Resources in Computing (JERIC)*, 5(3), Sept. 2005.

[2] R. Saikkonen, L. Malmi, and A. Korhonen, "Fully automatic assessment of programming exercises," ACM SIGCSE Bulletin Homepage, 33(3), pp. 133-136, Sept. 2001.

[3] M. Luck, M. Joy, "A secure on-line submission system," Software - Practice and Experience, 29 (8), pp. 721-740, 1999.

[4] D. Jackson, "A Semi-Automated Approach to Online Assessment," ACM SIGCSE Bulletin Homepage, 32(3), pp. 164-167, Sept. 2000.

[5] The ACM-ICPC International Collegiate Programming Contest Web Site sponsored by IBM. Available: http://cm2prod.baylor.edu/welcome.icpc.

[6] KDiff3 – Homepage. Available: http://kdiff3.sourceforge.net.

[7] JLex – A Lexical Analyzer Generator for Java$^{(TM)}$. Available: http://www.cs.princeton.edu/~appel/modern/java/JLex.

[8] CUP – LALR Parser Generator for Java. Available: http://www2.cs.tum.edu/projects/cup.

[9] The Lisp Programming Language. Available: http://www.engin.umd.umich.edu/CIS/course.des/cis400/lisp/lisp.html.

[10] A .V. Aho, R. Sethi, and J. D. Ullman 1986, "Compilers: Principles, Techniques, and Tools", Addison-Wesley, 164.

[11] The Plagiarism Checker. Available: http://www.dustball.com/cs/plagiarism.checker.

[12] Plagiarismdetect. Available: http://www.plagiarismdetect.com.

[13] The Scheme Programming Language. Available: http://groups.csail.mit.edu/mac/projects/scheme.

[14] Revised^5 Report on the Algorithmic Language Scheme. Available: http://schemers.org/Documents/Standards/R5RS.

Author:

Omar Mohammad Othman
Department of Computer Science and Engineering
Faculty of Media Engineering and Technology
German University in Cairo
New Cairo City, Cairo 11835, Egypt