An Educational Programming Environment for Novices

M. Satratzemi[1], V. Dagdilelis[2], G. Evangelidis[1], and V. Efopoulos[1]

[1] Department of Applied Informatics
[2] Department of Educational and Social Policy
University of Macedonia, Egnatia 156, GR-54006, GREECE

Abstract. In this paper we present an integrated educational programming environment, called X-Compiler, designed to introduce students to programming. This environment is based on a simple programming language (and a corresponding minimal assembly language) called X. The design and development of X-Compiler was based on the research findings regarding the problems met by novice programmers and the results of empirical studies that discover and study the mistakes they make on a regular basis. We first review the various categories of programming environments already developed to aid the teaching of programming to novices and we present the design principles we used for X-Compiler. Then, we describe our system in detail. X-Compiler offers interesting didactic features. Users get detailed feedback on the errors encountered during compilation, and are always aware of everything that happens during program execution (by seeing the correspondence between source and assembly code, the intermediate values of the machine registers, the system generated temporary variables, their own variables, and the contents of the output window). Moreover, users can alter the produced assembly code and then execute it. We offer a great number of detailed and explanatory messages to guide novice programmers when debugging their programs and help them write better programs.

1 Introduction

A great number of educational programming environments designed to support the teaching of programming to novices have been proposed and developed during the past years. These environments are based on the findings of the research community on the difficulties novice programmers encounter and empirical studies on the most common and persistent errors novices make. These programming environments can be categorized as follows:

Microworlds - Programming Mini-languages. The basic idea behind microworlds and mini-languages [4] is the use of a simple programming language to support the early stages of programming learning. The majority of programming microworlds integrates an environment based on a real life analogy and an actor that "lives" in this environment. A representative of this category is mini-language Karel the Robot [13] and its associated programming environment Karel Genie.

Compilers with advanced error-reporting capabilities. Many researchers support that the frustration experienced by many students when introduced to programming is attributed more to the programming environment used and less to the programming language itself. A representative example of a programming environment of this category is THETIS [8] that consists of an ANSI C compiler with a user interface that provides easy to use code development, debugging and visualization tools.

Visual Programming Languages. Researchers that support the use of visual languages [3] propose that the teaching of programming to novices should be based on an environment where algorithms can be expressed visually (for example with flow charts) rather than textually (with the use of a traditional text editor). The first interesting case of a visual language was BACCII [5].

Program Visualization Systems. Program visualization refers to the visualization of the code or the data structures of a program. It can be static (for example, when using a diagram to demonstrate the operation of a linked list) or dynamic (e.g., by highlighting the line of the source code being currently executed). A classic example of this category is DYNALAB [3].

In this paper we present our own simple programming language called X and its associated programming environment called X-Compiler that we designed and implemented as part of the pilot project "DELYS"[1]. We opted for a new environment because, (a) we wanted to design a programming environment specifically tailored for the needs of the Greek educational system, and (b) we intended to include features that the current research reports can benefit novice programmers. In Section 2 we discuss the principles behind the design of X and X-Compiler. Section 3 presents the specifications of X and its associated pseudo-assembly language. We show how each statement of X is translated into assembly code in Section 4. In Section 5 we describe the programming environment for X, and we conclude with Section 6.

2 Design Principles for X and X-Compiler

Programming environments for novice programmers should primarily be effective tools for achieving certain didactic goals, and secondarily innovative and state-of-the-art pieces of software. Our design was based on and influenced by the studies on the difficulties novice programmers encounter. Below, we list a number of principles [12] we considered essential while developing the programming environment for X.

Minimalism. The programming language should be as simple as possible. The use of types should be avoided and variables should not have to be declared before use [11]. Also, the programming environment should not present unnecessary information.

Functional and Syntactic Simplicity. Novice programmers are asked to program a mental machine. This is the machine whose nature is determined, or better, implied by the programming language. It is essential that the mental machine is as simple as possible, i.e., it should consist of a small number of components that interact in a well-defined and clear manner [7, 10, 14]. The mental machine should be characterized by (a) functional simplicity, that is, each command should be described by a simple and small set of machine functions, and (b) syntactic simplicity, that is, the

[1] This research was funded by the European Union and the Greek Ministry of Education as part of the project "EPEAEK: Operational Program for Education and Initial Occupational Training", B' CFS, Pilot Project "ODYSSEIA: Development of Educational Software Accompanied by Pedagogical and Technological Documentation".

program	`BEGIN` { statement`;`}* `END.`
statement	id `:=` expr | `READ` id | `WRITE` expr | `IF` relexpr `THEN` statement | `WHILE` relexpr `DO` statement | `BEGIN` {statement`;`}* `END`
id	*any string consisting of letters, digits and underscore and starting with a letter*
expr	id | number | expr op expr | `(`expr`)`
op	$+ \mid - \mid * \mid /$
number	*any long integer*
relexpr	expr relop expr
relop	$> \mid = \mid <>$
comments	*anything enclosed in curly brackets*

Table 1. Specification of X

syntactic rules of the programming language should be consistent and should not have special cases and/or exceptions.

Stepped execution and control through visual feedback. Instant feedback can help novice programmers implement and debug their programs. A graphical debugger is useful even for correct programs: it can help novice programmers understand the way their programs work. A programming environment should help novice programmers test, debug, and execute their programs [6]. It is essential that the programming environment include a low-level debugger and a code execution tracer together with data visualization [8, 15]. Commercial programming environments hide the details of program execution and students tend to conceive program execution as an input/output process [4] failing to realize the semantics of the programming language.

Extensive debugging and programming environment usage help. The programming environment should provide novice programmers with many detailed and explanatory messages that can guide them when debugging their programs. Error messages should be expressed in a natural language and the programming environment should propose ways to correct the errors [15]. Well-designed systems are intuitive and users do not need a manual to use them. Of course, provision of on-line help or tutorials [2] is essential and helpful, but these should be minimal and clear.

3 Languages X and pseudo-assembly

We have designed a Pascal-like language, called X. The language supports the assignment, if ...then, while ...do, read, write and compound statements. Identifiers and numbers are integers and all, possibly nested, arithmetic expressions evaluate to integers. X supports only three relational operators: $>, =,$ and $<>$. A comment is text enclosed in curly brackets ({}). Table 1 shows the full specification of X. In Figure 1, we give the X code for computing the factorial of an integer.

The assembly language used is a pseudo-assembly that runs on a virtual machine with two registers and includes the basic LOAD, STORE, COMPARE, JUMP, ADD,

```
{ computing the factorial of an integer x }
begin
    read x;
    i := 1;
    factorial := 1;
    while x > i do
    begin
        i := i + 1;
        factorial := factorial * i;
    end;
    write factorial;
end.
```

Fig. 1. A sample X program

etc., instructions needed to implement the source language. A detailed description of the instructions of the pseudo-assembly can be found in Table 2 in the Appendix.

4 Compiling X to pseudo-assembly

Every statement of X is translated into one or more pseudo-assembly statements. In Table 3 in the Appendix you can see the way each generic X statement is translated (or compiled). The result of the evaluation of all arithmetic and relational expressions is stored in register 0. Temporary variables may be used during the computation of expressions. Names of the type cond_mid_N, while_begin_N, tempr_vrbl_N, where (N=1,2,3,...), refer to the N^{th} conditional statement, the N^{th} while statement, and the N^{th} temporary variable respectively.

In Figure 2 we give the assembly code X-Compiler produces for the sample X program of Figure 1. We also demonstrate the way each X statement is translated into its equivalent pseudo-assembly code.

The X-Compiler programming environment has been implemented on the Microsoft Windows platform using Macromedia Director 7 and the compiler construction tools LEX and YACC [1].

X-Compiler allows users to edit, debug, and execute their programs. It consists of five windows (1 - source code, 2 - assembly code, 3 - system registers & temporary variables, 4 - user variables, 5 - output), and has two modes of operation (novice and advanced) (see Figure 3). In novice mode, only windows 1 and 5 are active, whereas in advanced mode all windows are active. Of course, users can activate or deactivate any window any time.

The provided menu-bar and window-specific toolbars allow the intuitive use of the programming environment (open, save, and edit source or assembly code, compile, execute, or step-execute either type of code, arrange windows, get help on the operation of the programming environment or the X-language).

We took extra care to include a versatile help system both for the programming environment and X, since the current research reports many problems in this area [8].

```
begin
                                        BLOCK factorial
                                        BLOCK x
                                        BLOCK i
   read x;                              READ(0)
                                        STORE(0,x)
   i := 1;                              LOAD_IMMEDIATE(0,1)
                                        STORE(0,i)
   factorial := 1;                      LOAD_IMMEDIATE(0,1)
                                        STORE(0,factorial)
   while x > i do                    while_begin_1:
   begin                                LOAD(0,x)
                                        LOAD(1,i)
                                        COMPARE(0,1,0)
                                        JUMP_NEGATIVE(0,while_end_1)
      i := i + 1;                       LOAD(0,i)
                                        LOAD_IMMEDIATE(1,1)
                                        ADD_R(0,1,0)
                                        STORE(0,i)
      factorial := factorial * i;       LOAD(0,factorial)
                                        LOAD(1,i)
                                        MUL_R(0,1,0)
                                        STORE(0,factorial)
   end;                                 JUMP(while_begin_1)
   write factorial;                  while_end_1:
                                        LOAD(0,factorial)
                                        WRITE(0)
end.
```

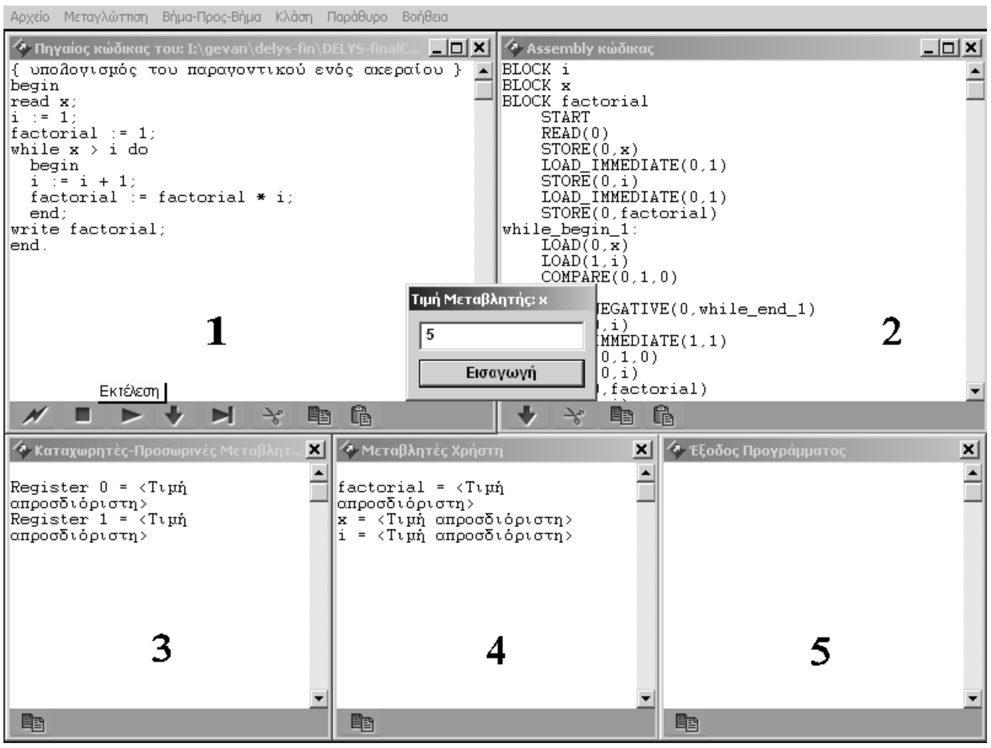
Fig. 2. The correspondence of X and assembly code for computing factorial

We support (a) regular help files for the programming environment (Figure 4), and (b) an editor sensitive to user double-clicks, i.e., users can double-click keywords, operators or delimiters to obtain detailed information about them (Figure 5a).

During compilation, syntactic errors in the source code trigger a pop-up window that contains two drop-down lists, one for the detected errors and one for the warnings issued by the compiler. Users can choose the list element they desire to get an explanation of the type of the error or warning. At the same time the appropriate line of the source code is highlighted (Figure 5b).

Once users succeed in compiling their code they can either execute it or step-execute it and examine what actually happens during execution. For each source code statement the corresponding assembly code statement(s) are highlighted and at the same time the appropriate system registers, temporary variables, and user variables get updated if necessary (see Figure 6).

Input statements are handled by using a pop-up window that allows users to enter the desired value for their integer variables (see in the center of Figure 3). The output window displays the output generated by the WRITE statements of the user programs.



```
{ υπολογισμός του παραγοντικού ενός ακεραίου }
begin
read x;
i := 1;
factorial := 1;
while x > i do
  begin
  i := i + 1;
  factorial := factorial * i;
  end;
write factorial;
end.
```

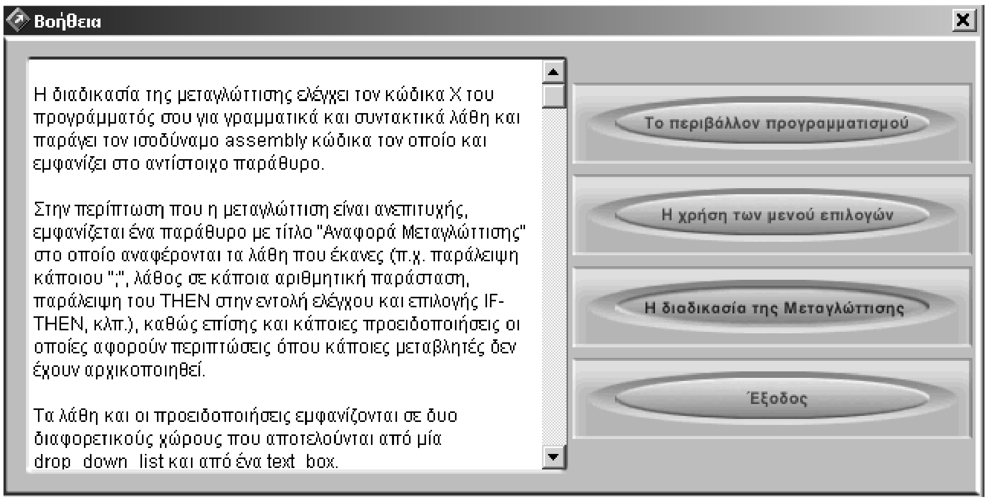
```
BLOCK i
BLOCK x
BLOCK factorial
    START
    READ(0)
    STORE(0,x)
    LOAD_IMMEDIATE(0,1)
    STORE(0,i)
    LOAD_IMMEDIATE(0,1)
    STORE(0,factorial)
while_begin_1:
    LOAD(0,x)
    LOAD(1,i)
    COMPARE(0,1,0)

          EGATIVE(0,while_end_1)
          ,i)
          MMEDIATE(1,1)
          0,1,0)
          0,i)
          ,factorial)
```

5

```
Register 0 = <Τιμή
απροσδιόριστη>
Register 1 = <Τιμή
απροσδιόριστη>
```

```
factorial = <Τιμή
απροσδιόριστη>
x = <Τιμή απροσδιόριστη>
i = <Τιμή απροσδιόριστη>
```

Fig. 3. The X-Compiler programming environment



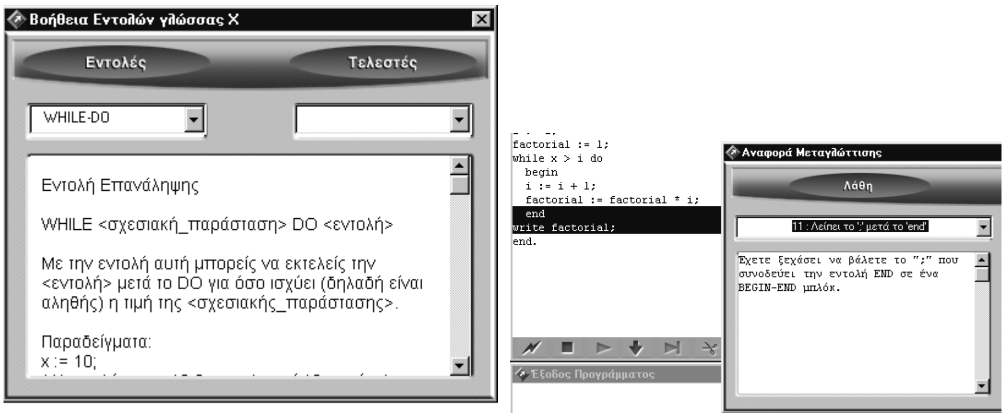
Βοήθεια
Το περιβάλλον προγραμματισμού
Η χρήση των μενού επιλογών
Η διαδικασία της Μεταγλώττισης
Έξοδος

Η διαδικασία της μεταγλώττισης ελέγχει τον κώδικα X του προγράμματός σου για γραμματικά και συντακτικά λάθη και παράγει τον ισοδύναμο assembly κώδικα τον οποίο και εμφανίζει στο αντίστοιχο παράθυρο.

Στην περίπτωση που η μεταγλώττιση είναι ανεπιτυχής, εμφανίζεται ένα παράθυρο με τίτλο "Αναφορά Μεταγλώττισης" στο οποίο αναφέρονται τα λάθη που έκανες (π.χ. παράλειψη κάποιου ";", λάθος σε κάποια αριθμητική παράσταση, παράλειψη του THEN στην εντολή ελέγχου και επιλογής IF-THEN, κλπ.), καθώς επίσης και κάποιες προειδοποιήσεις οι οποίες αφορούν περιπτώσεις όπου κάποιες μεταβλητές δεν έχουν αρχικοποιηθεί.

Τα λάθη και οι προειδοποιήσεις εμφανίζονται σε δυο διαφορετικούς χώρους που αποτελούνται από μία drop down list και από ένα text box.

Fig. 4. The help system of X-Compiler



Εντολή Επανάληψης

WHILE <σχεσιακή_παράσταση> DO <εντολή>

Με την εντολή αυτή μπορείς να εκτελείς την <εντολή> μετά το DO για όσο ισχύει (δηλαδή είναι αληθής) η τιμή της <σχεσιακής_παράστασης>.

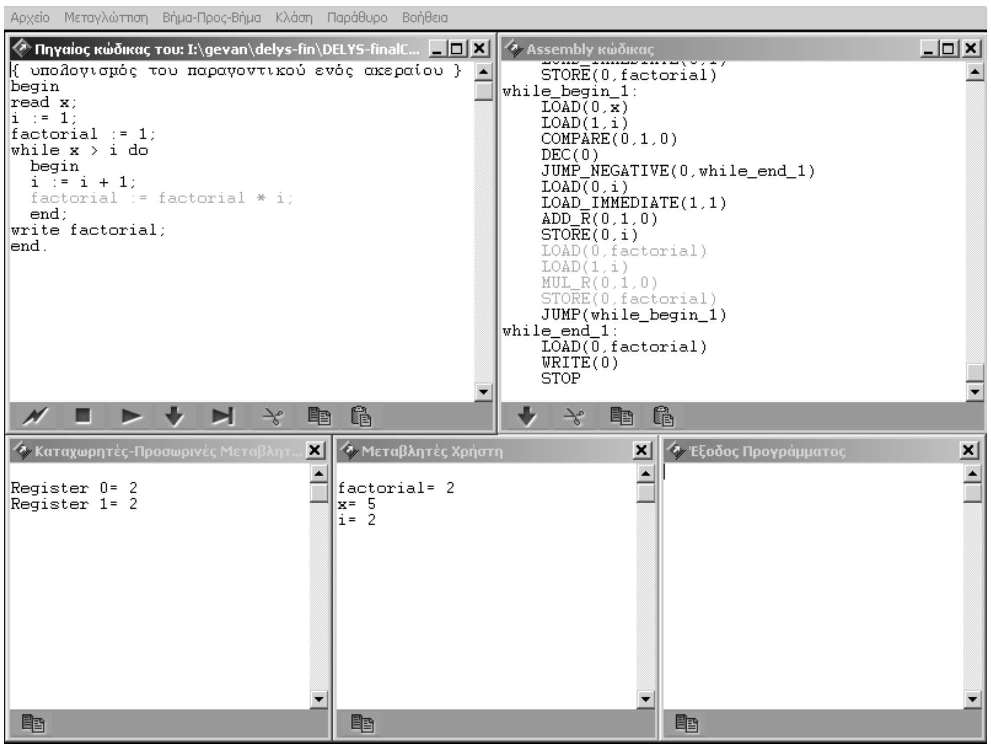
Παραδείγματα:
x := 10;

```
factorial := 1;
while x > i do
  begin
  i := i + 1;
  factorial := factorial * i;
  end
write factorial;
end.
```

11 : Λείπει το ';' μετά το 'end'

Έχετε ξεχάσει να βάλετε το ";" που συνοδεύει την εντολή END σε ένα BEGIN-END μπλόκ.

Fig. 5. (a) On-line help for X statements, (b) Error detection and advice to the programmer


Αρχείο Μεταγλώττιση Βήμα-Προς-Βήμα Κλάση Παράθυρο Βοήθεια
Πηγαίος κώδικας του: I:\gevan\delys-fin\DELYS-finalC...
Assembly κώδικας
Καταχωρητές-Προσωρινές Μεταβλητ...
Μεταβλητές Χρήστη
Έξοδος Προγράμματος

```
{ υπολογισμός του παραγοντικού ενός ακεραίου }
begin
read x;
i := 1;
factorial := 1;
while x > i do
  begin
  i := i + 1;
  factorial := factorial * i;
  end;
write factorial;
end.
```

```
    STORE(0,factorial)
while_begin_1:
    LOAD(0,x)
    LOAD(1,i)
    COMPARE(0,1,0)
    DEC(0)
    JUMP_NEGATIVE(0,while_end_1)
    LOAD(0,i)
    LOAD_IMMEDIATE(1,1)
    ADD_R(0,1,0)
    STORE(0,i)
    LOAD(0,factorial)
    LOAD(1,i)
    MUL_R(0,1,0)
    STORE(0,factorial)
    JUMP(while_begin_1)
while_end_1:
    LOAD(0,factorial)
    WRITE(0)
    STOP
```

```
Register 0= 2
Register 1= 2
```

```
factorial= 2
x= 5
i= 2
```

Fig. 6. Step execution of X and pseudo-assembly

An interesting feature of the assembly code window is the ability to edit/alter the compiler produced assembly code that corresponds to a given source code fragment and execute it. Since the compiler produces non-optimized assembly code this feature can allow teachers guide their students in manually optimizing their assembly code. Alternatively, users can write their assembly programs from scratch.

5 Summary

During project development, students of three Greek high schools tested X-Compiler and their remarks and suggestions were taken into account. X-Compiler is about to be used by the Greek Ministry of Education in a number of secondary education schools and in the entry-level university courses on programming we teach (especially its assembly language features).

Currently, we are in the process of implementing some additions to the software concerning, (a) a small extension of X to include strings and procedures, and (b) the creation of a "smart" advisor on the logical errors made by students.

Appendix

instruction	explanation
`BLOCK v`	Declare an integer variable or a memory position with name v
`LOAD(r, v)`	store contents of memory location v in register r (r can be 0 or 1)
`LOADN(r, n)`	store number n in register r
`STORE(r, v)`	store contents of register r to memory location v

`ADD_R(r1, r2, r3)`
`SUB_R(r1, r2, r3)`
`MUL_R(r1, r2, r3)`
`DIV_R(r1, r2, r3)`
	add/subtract/multiply/divide contents of registers $r1$ and $r2$
	and store the result in register $r3$

`COMPARE(r1, r2, r3)`	compare the contents of registers $r1$ and $r2$ and store the result in register $r3$; the result is -1 if $r1 < r2$, 1 if $r1 > r2$, and 0 if $r1 = r2$
`INC(r)`	increment the contents of register r
`DEC(r)`	decrement the contents of register r
`NEG(r)`	negate the contents of register r
`my_label:`	declare a label with the name my_label
`JUMP_ZERO(r, lb)`	jump to lb if contents of $r = 0$
`JUMP_NEGATIVE(r,lb)`	jump to lb if contents of $r < 0$
`JUMP_POSITIVE(r, lb)`	jump to lb if contents of $r > 0$
`JUMP(lb)`	unconditionally jump to lb
`READ(r)`	store user input to register r
`WRITE(r)`	print contents of register r to the output window

Table 2. The pseudo-assembly used in X-Compiler

X statement	Equivalent assembly code
`write` <arithmetic_expr>	<code for the arithmentic_expr>
	`WRITE(0)`
`read` <id>	`READ(0)`
	`STORE(0,` <id>`)`
`if` <relational_expr> `then` <stmt>	<code for relational_expr>
	`JUMP_cond`[2]`(0, cond_mid_N)`
	<code for stmt>
	`cond_mid_N:`
`while` <relational_expr> `do` <stmt>	`while_begin_N:`
	<code for relational_expr>
	`JUMP_cond(0, while_end_N)`
	<code for stmt>
	`JUMP(while_begin_N)`
	`while_end_N:`
<id> `:=` <arithmetic_expr>	<code for arithmetic_expr>
	`STORE(0,` <id>`)`
<arithmetic_expr1> op[3]<arithmetic_expr2>	<code for arithmetic_expr2>
	`STORE(0, tempr_vrbl_N)`
	<code for arithmetic_expr1>
	`LOAD(1, tempr_vrbl_N)`
	`ACTION`[4]`(0, 1, 0)`
`-` <arithmetic_expr>	<code for arithmetic_expr>
	`NEG(0)`
(<arithmetic_expr>)	<code for arithmetic_expr>
<id>	`LOAD(0,` <id>`)`
<number>	`LOAD_IMMEDIATE(0,` <number>`)`
<arithmetic_expr1> rel_op[5]<arithmetic_expr2>	<code for arithmetic_expr2>
	`STORE(0, tempr_vrbl_N)`
	<code for arithmetic_expr1>
	`LOAD(1, tempr_vrbl_N)`
	`COMPARE(0, 1, 0)`

Table 3. The pseudo-assembly used in X-Compiler

References

1. A.V. Aho, R. Sethi, and J.D. Ullman, "Compilers: principles, techniques, tools", Addison-Wesley, 1988.
2. B. Bell, J. Rieman, and C. Lewis, "Usability Testing of a Graphical Programming System: Things We Missed in a Programming Walkthrough". *In Proceedings of ACM CHI '91 Conference on Human Factors in Computing Systems*, pp. 7-12, 1991.

[2] where `JUMP_cond` is one of `JUMP_ZERO, JUMP_POSITIVE, JUMP_NEGATIVE`

[3] where op is `+, -, *, /`

[4] where `ACTION` is `ADD_R, SUB_R, MUL_R  DIV_R`

[5] where relational_op is `>, <>, =`

3. M. Birch, C. Boroni, F. Goosey, S. Patton, D. Poole, C. Pratt, and R. Ross, “DYNALAB: A Dynamic Computer Science Laboratory Infrastructure Featuring Program Animation”, *ACM SIGSCE Bulletin*, pp. 29-33, 1995.
4. P. Brusilovsky et al, “Mini-languages: a way to learn programming principle”, *Education and Information Technologies*, 2, 65-83, 1997.
5. B. Calloni and D. Bagert, “Iconic Programming in BACCII vs. Textual Programming: which is a better learning environment?”, *ACM, SIGSCE ’94*, Phoenix AZ, pp. 188-192, 1994.
6. C. DiGiano, R. Baecker, and A. Marcus, “Software visualization for Debugging”, *Communications of the ACM*, Vol. 40, No. 4, pp. 44-54, 1997.
7. B. Du Boulay, T. O’Shea, and J. Monk, “The Black Box Inside the Glass Box: Presenting Computing Concepts to Novices”, *Studying The Novice Programmer*, E. Soloway and J. Sprohrer (Eds.), Lawrence Erlbaum Associates, pp. 431-446, 1989.
8. S.N. Freund and E.S. Roberts, “THETIS: An ANSI C programming environment designed for introductory use”, *ACM SIGSCE ’96*, Philadelphia, PA, USA, pp. 300-304, 1996.
9. MacGNOME Project, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213.
10. P. Mendelsohn, T.R.G. Green, and P. Brna, “Programming Languages in Education: The Search for an Easy Start”, *Psychology of Programming*, J. Hoc, T. Green, R. Samurcay, and D. Gilmore (Eds.), Academic Press, 175-200, 1990.
11. P. Miller, J. Pane, G. Meter, and S. Vorthmann, “Evolution of Novice Programming Environments: the Structure Editors of Carnegie Mellon University”, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213-3890, 1996.
12. J.F. Pane and B.A. Myers, “Usability Issues in the Design of Novice Programming Systems”, Technical Report CMU-CS-96-132, School of Computer Science, Carnegie Mellon University, 1996.
13. R.E. Pattis, J. Roberts, and M. Stehlik, “Karel - The Robot, A Gentle Introduction to the Art of Programming”, 2nd edn., New York, Wiley, 1995.
14. M. Ruckert and R. Halpern, “Educational C”, *ACM SIGSCE Bulletin*, pp. 6-9, 1993.
15. T. Schorsch, “CAP: An Automated self-assessment to tool to check Pascal programs for syntax, logic and style errors”, *ACM SIGCSE ’95*, pp.168-172, 1995.