International Science Index, Computer and Information Engineering Vol:6, No:7, 2012 waset.org/Publication/5580

EGCL: An Extended G-Code Language with Flow Control, Functions and Mnemonic Variables

Oscar E. Ruiz, S. Arroyave, J. F. Cardona

Abstract—In the context of computer numerical control (CNC) and computer aided manufacturing (CAM), the capabilities of programming languages such as symbolic and intuitive programming, program portability and geometrical portfolio have special importance. They allow to save time and to avoid errors during part programming and permit code re-usage. Our updated literature review indicates that the current state of art presents voids in parametric programming, program portability and programming flexibility. In response to this situation, this article presents a compiler implementation for EGCL (Extended G-code Language), a new, enriched CNC programming language which allows the use of descriptive variable names, geometrical functions and flow-control statements (if-then-else, while). Our compiler produces low-level generic, elementary ISO-compliant G-code, thus allowing for flexibility in the choice of the executing CNC machine and in portability. Our results show that readable variable names and flow control statements allow a simplified and intuitive part programming and permit re-usage of the programs. Future work includes allowing the programmer to define own functions in terms of EGCL, in contrast to the current status of having them as library built-in functions.

Keywords—CNC Programming, Compiler, G-code Language, Numerically Controlled Machine-Tools.

Glossary

APT	: Automatically Programming Tool
AST	: Abstract Syntax Tree
EGCL	: Extended G-Code Language
IGCL	: ISO G-code Language
GUI	: Graphic User Interface
NC	: Numerical Control
NCPP	: Numerical Control Program Processor
STEP-NC	: Standard for the Exchange of Product data for NC

I. Introduction

THE interaction between programmers and CNC machine-tools is implemented through machining languages. The most widely used language is G-code, even though there are other programming languages such as APT and STEP-NC ([11], [6] see section II-A1).

G-code was defined in the 1960's by the ISO6983 standard ([5]). It can be considered as a low-level programming language as it lacks implementation variables, arithmetic and boolean expressions, flow control structures and function calls. There are currently several ways to produce G-code. They include: (1) manual programming, (2) vendor macro-language programming and (3) CAD-CAM systems. Manual programming, covering most industry applications ([9]), is a technique based on the ISO G-code language (IGCL), in which each movement through coordinate point sequence must be specified by an instruction. Additionally, the traditional G-code language does not permit symbolic coordinate programming. This in general implies reprogramming and new programming errors when changes in the designed part are made. At the expense of program portability, NC machine-tools vendors have extended the IGCL creating their own macro-languages by adding some features such as numeric variables, flow control statements and machining cycles (e.g. drilling cycle). This implies that each vendor macro-language can only be executed on a specific machine-tool.

Prof. Oscar E. Ruiz is the Coordinator of the CAD/CAM/CAE Laboratory at Universidad EAFIT, Medellín, Colombia. Email: oruiz@eafit.edu.co
S. Arroyave is research assistants with the CAD/CAM/CAE Laboratory at Universidad EAFIT, Medellín, Colombia. Email: sarroyav@eafit.edu.co
Prof. J. F. Cardona is with the Department of Compuer Science of Universidad EAFIT, Medellín, Colombia. Email: fcardona@eafit.edu.co

In CAD-CAM systems, machining instructions are obtained from a 3D geometric model and user-defined machining strategies, obtaining a specific G-code program by selecting a post-processor of a given machine vendor.

The APT programming language was developed in the 1950's at MIT ([11]). In APT, the user must describe the geometrical part and give a high-level description of the tool path. Notice that in CNC-oriented CAD-CAM systems, the geometry of the part is either imported as a file or defined via a GUI and the tool path is defined by expert systems which propose one of several machining programmer-friendly strategies.

In the present article, the problem of IGCL insufficient programming capabilities is faced by defining a new language and by creating its compiler to produce generic ISO G-code, thus allowing for flexibility in the choice of the executing CNC machine and in code re-usability.

This paper is organized as follows: In section II, a literature review is presented. Section III explains how the grammar was defined and the proposed language compiler was implemented. Section IV discusses the results and application cases. Section V concludes the article and poses future work questions.

II. Literature Review

Many discussions take place on how to improve the capabilities of IGCL programming and on how to solve portability problems of vendor programs. Some authors face this problem by creating new languages or by modifying the machine control systems whereas other propose to translate G-code programs from a specific vendor language to another. In this

International Science Index, Computer and Information Engineering Vol:6, No:7, 2012 waset.org/Publication/5580

section we discuss the existing literature, giving a taxonomy along with conclusions about the aspects to improve, which explain the objectives of the present article.

A. CNC Languages

1) STEP-NC: It is a programming language specified in 2003 by the ISO14649 standard ([6]). This language does not only allow storing machining instructions, but also relevant information as the geometrical description of a part, work plans, machining strategies, tools and so on ([4]). STEP-NC was created to improve the G-code. Nevertheless, replacing the traditional way to program CNC will take many years due to the costs of the technological transformation of the industry ([13]). In addition, distinguished CNC machine-tool vendors (e.g. Fanuc and Mitsubishi) have not yet implemented STEP-NC in their products ([3]). On the other hand, STEP-NC is designed to be used by integrated complete CAD-CAM systems, resulting non affordable for small companies ([12]).

2) G-code Expansions: Neagle and Wiegley ([10]) report the embedding of IGCL grammar into Java Language. The user can then program G-code instructions inside a Java script and compile it to produce traditional G-code. This implies that the compiler generates the Abstract Syntax Tree (AST) as an intermediate representation of the source program, solely in terms of Java language. It does not allow, before the code generation, to handle the AST to do a code optimization or a geometrical tool path verification.

Arroyo, Ochoa, Silva y Vidal ([2]) propose the use of Haskell to design IGCL programs. Haskell is a high level, general-purpose and pure functional programming language. The authors create functions in Haskell which produce G-Code instructions. In order to produce basic G-code, the programmer must execute the functions. This implies to assemble these instructions with the rest of the code, resulting in an error-prone process.

B. Program Processors

In a CNC machine, the NC program processor (NCPP) is in charge of checking the syntax of NC programs and decoding them into specific outputs such as motion command, parameter setting or error messages ([3]). Some authors have created new proccesors in order to solve IGCL shortcoming. References [3] and [9] present a universal NCPP that accepts NC programs inputs in different vendor macro languages such as Fanuc and Mitsubishi. Reference [15] proposes an intelligent open CNC system in which natural language (e.g. English) is used to describe the machining routines. This kind of solution could be undesirable for companies because they must modify the existing control system by including the new NCPP. It must also be noticed that natural language technology still presents inherent unsolved ambiguities an challenges, not only related to CNC activities.

C. Translators

In order to solve the portability problems of IGCL programs, Schroeder and Hoffman ([12]) proposed a translator that can take a program written in a specific vendor language and convert it into another specific vendor language by defining the converting grammar rules. This proposal, being a specific solution for portability problems instead of a general one, requires that the user have knowledge in formal language theory to define the converting grammar rules. In addition, it is a language-to-language converter. So there could potentially be $2.n.(n-1)$ converters for a set of n languages.

D. Conclusions of the Literature Review

According to the taxonomy conducted in this literature review, there are several capabilities that remain elusive in CNC programming. These include: (1) program portability (2) mnemonic variable names, (3) symbolic coordinates programming and (4) flow control structures such as IF and WHILE.

In response to these limitations, this paper presents the implementation of EGCL, an Extended G-Code Language for CNC machining, which enhances the traditional IGCL by adding to it the grammar of arithmetic and boolean expressions and control flow statements. By taking advantage of the fact that most machining controls are able to process IGCL programs, the output of our EGCL compiler is a program written in vendor-independent IGCL. This solution is available without the need of replacing or modifying the machine control.

III. Methodology

A. Compiler Design

In order to design a extended G-code language (EGCL) able to recognize readable variable names, symbolic coordinates, IF and WHILE statements and functions, it was necessary to define a non-ambiguous grammar which accepts IGCL, arithmetic and boolean expressions and control flow statements grammars.

The front-end of a compiler is composed by the lexical analyzer (also called lexer) and the syntactical analyzer (also called parser). They have the responsibility of construct an intermediate representation (AST) of the input program (see Figure 1). The back-end is in charge of code generation. The middle-end is responsible of performing transformations on the intermediate representation, so that the back end can produce a better target program than it would otherwise have been produced from an unoptimized intermediate representation ([1]).

1) Lexical Analyzer: In our case, the lexical analyzer was created with the software Flex©, a generator of lexical scanners ([8]). The finite automata regular definitions for lexical analyzer are shown in Figure 2.

2) Syntactic Analyzer: The syntactic analyzer or parser determines whether a sequence of valid EGCL words (tokens) corresponds indeed to a valid EGCL sentence or sequence of sentences. A parsing process does not perform semantic, geometric and machining validations. The semantic process is in charge of such check ups. In the parser, if an unexpected token is read, the parser stops the compilation and shows an error message. On the other hand, if a valid statement or sequence of statements are recognized by the parser, it stores

International Science Index, Computer and Information Engineering Vol:6, No:7, 2012 waset.org/Publication/5580

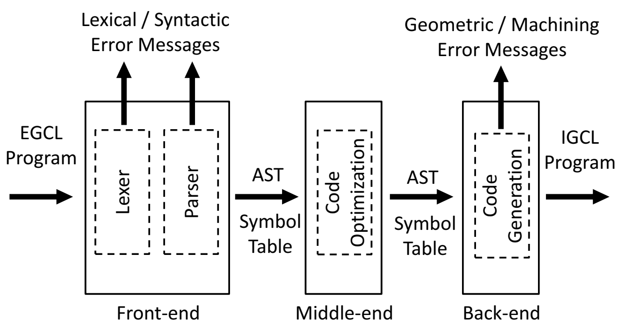

Fig. 1. Internal scheme of the compiler.

LETTER	→	`[a-zA-Z]`
DIGIT	→	`[0-9]`
POSINTEGER	→	`0` | `[1-9]` *DIGIT**
POSREAL	→	*POSINTEGER*? . *DIGIT**
INTEGER	→	-? *POSINTEGER*
REAL	→	-? *POSREAL*
PROG_NAME	→	`(O | o)` $DIGIT^+$
LINE_NUM	→	`(N | n)` $DIGIT^+$
G0_CODE	→	`(G | g) 0? 0`
G1_CODE	→	`(G | g) 0? 1`
G2_CODE	→	`(G | g) 0? 2`
G3_CODE	→	`(G | g) 0? 3`
X_COORD	→	`(X | x)` (*REAL* | *INTEGER*)
Y_COORD	→	`(Y | y)` (*REAL* | *INTEGER*)
Z_COORD	→	`(Z | z)` (*REAL* | *INTEGER*)
X_CENTER_COORD	→	`(I | i)` (*REAL* | *INTEGER*)
Y_CENTER_COORD	→	`(J | j)` (*REAL* | *INTEGER*)
Z_CENTER_COORD	→	`(K | k)` (*REAL* | *INTEGER*)
RADIUS	→	`(R | r)` (*POSREAL* | *POSINTEGER*)
ADD_INFO	→	*M_FUN* | *G_FUN* | *TOOL_COMP*
SPEED	→	*FEED_RATE* | *SPINDLE_SPEED*
T_CHANGE_CODE	→	`(M | m) 0? 6`
TOOL_NUM	→	`(T | t)` $DIGIT^+$
TOOL_COMP	→	`(G | g)` (41 |42)
G43_CODE	→	`(G | g)` (43)
LENGTH_ADD	→	`(H | h)` $DIGIT^+$
FEED_RATE	→	`(F | f)` (*POSREAL* | *POSINTEGER*)
SPINDLE_SPEED	→	`(S | s)` (*POSREAL* | *POSINTEGER*)
M_FUN	→	`(M | m)` $DIGIT^+$
G_FUN	→	`(G | g)` $DIGIT^+$
FINISH	→	`(M | m)` (30 | 02 | 2)
TRUE	→	`true`
FALSE	→	`false`
ELSEIF	→	`else if`
IF	→	`if`
ELSE	→	`else`
WHILE	→	`while`
SPIRAL	→	`spiral`
HOLESLINE	→	`holesLine`
CIRCARRAY	→	`circArray`
TOK_X	→	X | x
TOK_Y	→	Y | y
TOK_Z	→	Z | z
IDENT	→	*LETTER* (*DIGIT* | *LETTER* | _)*
BINOP	→	+ | - | $\star$ | \ | ^
BOOLOP	→	&& | ||
UNIOP	→	!
LOGOP	→	== | != | <= | < | >= | >

Fig. 2. Regular definitions for the Lexical Analyzer EGCL.

prog	→	ϵ | *start stms end*
start	→	*PROG_NAME*
end	→	*FINISH*
stms	→	ϵ | *stms stm*
stm	→	*cond* | *loop* | *g_line* | *functions*
g_line	→	*move_code add_info_l*
	|	`T_CHANGE_CODE TOOL_NUM`
	→	`G43_CODE` *z_coord* `LENGTH_ADD`
move_code	→	*coord_l*
	|	`G0_CODE` *coord_l*
	|	`G1_CODE` *coord_l*
	|	*coord_l cen_or_rad*
	|	`G2_CODE` *coord_l cen_or_rad*
	|	`G3_CODE` *coord_l cen_or_rad*
coord_l	→	`x_coord y_coord z_coord`
x_coord	→	`X_COORD` | *x_exp*
x_exp	→	`TOK_X` "<" *exp* ">"
y_coord	→	`Y_COORD` | *y_exp*
y_exp	→	`TOK_Y` "<" *exp* ">"
z_coord	→	`Z_COORD` | *z_exp*
z_exp	→	`TOK_Z` "<" *exp* ">"
exp	→	`TRUE` | `FALSE`
	|	`INTEGER` | `REAL` | `IDEN` | *call*
	|	*exp* `BINOP` *exp*
	|	"(" `exp` ")"
call	→	`IDENT` "(" *exp* ")"
assing	→	`IDENT` = *exp*
bool_exp	→	`TRUE` | `FALSE`
	|	*bool_exp* `BOOLOP` *bool_exp*
	|	*exp* `LOGOP` *exp*
	|	"(" *bool_exp* ")"
loop	→	`WHILE` "(" *bool_exp* ")" "{" *stms* "}"
cond	→	`IF` (*bool_exp* ")" "{" *stms* "}" *elseif else*
else	→	ϵ | `ELSE` "{" *stms* "}"
elseif	→	ϵ | `ELSEIF` "(" *bool_exp* ")" "{" *stms* "}"
functions	→	`SPIRAL_TOK` | `HOLES_TOK` | `C_ARRAY_TOK`

Fig. 3. Context-free grammar for the Syntactic Analyzer EGCL.

```
 1: O001;                              Program name
 2: Tool_num_1 = true;                 Variable declaration
 3: if (Tool_num_1 == true)
 4: {
 5:    M06 T01;                        To take tool number 1
 6: }
 7: else
 8: {
 9:    M06 T02;                        To take tool number 2
10: }
11: M30;                               Program end
```

Fig. 4. An IF-ELSE instruction in EGCL.

the sequence in a hierarchical structure of internal instructions, which is an intermediate representation of the input program. This intermediate representation is usually built as a tree data structure called Abstract Syntax Tree (AST). Figure 5 shows the AST produced for the EGCL code displayed in Figure 4.

The syntactic analyzer was created with the software Bison© ([7]), based on the grammar shown in Figure 3.

3) Code Generator: The AST is generated as a by-product of the syntactical validation of the input EGCL code. A further processing of the AST translates it into a IGCL file.

In line 2 of Figure 4, the assignment in EGCL expresses that the variable `Tool_num_1` must be created and a value assigned to it. If the boolean expression in line 3 evaluates to TRUE, the statement of the IF domain is executed and then the text `M06 T01` (tool change in the magazine of the CNC machine) is written to the output file. The output stream is ended by the closing `M30` text. The output file looks as shown in Figure 6. This simple example illustrates the compiling process showing the capabilities of the EGCL to use mnemonic

International Science Index, Computer and Information Engineering Vol:6, No:7, 2012 waset.org/Publication/5580

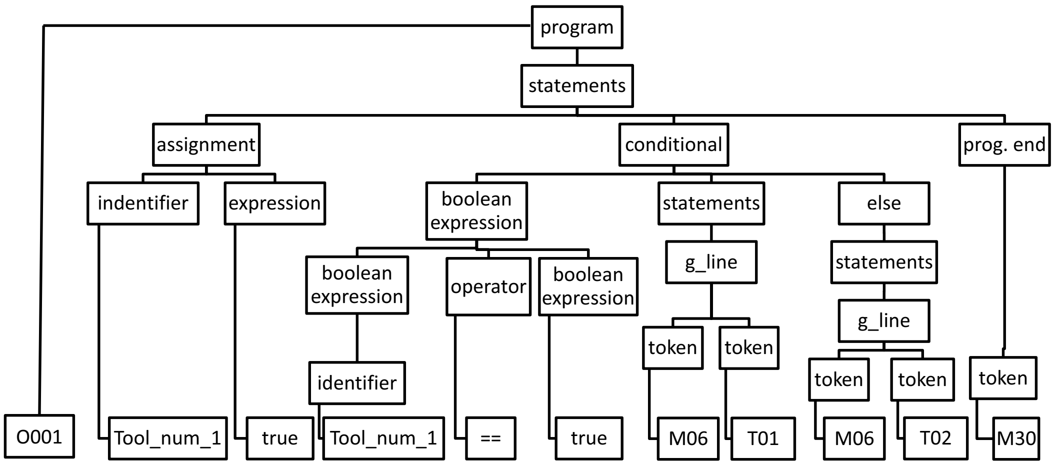

Fig. 5. AST for the EGCL example program in Figure 4.

```
1: O001
2: N10 M06 T01
3: N20 M30
```

Fig. 6. Output of IF-ELSE Instruction in EGCL in Figure 4

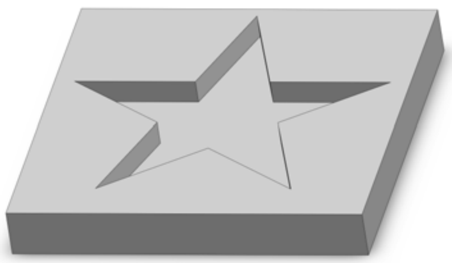

Fig. 7. Proposed Star Pocket part.

variables and to control the program flow. In this manner, one EGCL script is able to produce several different IGCL programs by changing some geometric parameters defined at the header. The produced IGCL programs will generate diverse parts once they are executed in a CNC machine.

4) Code Simulator: A tool path simulator based in OpenGL© was created in our work to check the output of EGCL compiler. This simulation is generated by other compiler whose input is an IGCL program and whose output is a graphical tool path representation. CncSimulator© ([14]) was used to double-check the EGCL compiler output and the tool path simulator.

IV. Results and Discussion

In order to check the functioning of the designed compiler, two application cases were proposed (Figures 7 and 11), as follows.

A. Star Pocket Part

In the metalwork industry, some parts must be machined using a contour-parallel tool path strategy. The programming of this type of parts can be simplified with the implementation of loop statements. In order to program the part proposed in Figure 7 which its details are shown in Figure 8, it was first defined some geometric parameters (lines 6-8, Figure 9). Two WHILE instructions are used in lines 23 y 25 respectively. The outer loop (line 23) controls the number of passes depending on the total of the hole depth (`Depth`) and the depth per pass (`DPass`). The innermost loop (line 25) controls the number of parallel cuts in one pass, depending on the tool diameter and the overlap percentage of the tool between parallel passes (`OV_LAP`) and the circumscribed star diameter in a pass (`DL`, see Figure 8). Finally, parametric statements of movements were programmed in lines 47-58 of Figure 9. A stream of elementary G-code machining instructions were obtained as the result of the EGCL program compilation. They were simulated in the software CncSimulator© ([14]) obtaining the result shown in Figure 10.

This application case highlights the capabilities of the EGCL in geometry parametrization through symbolic programming and the advantages of nested loops statements to control the number of passes and parallel cuts.

B. Drill Target Pattern Part

In order to illustrate the capabilities of the language to program hole drilling, a part with a large number of holes was designed (see Figures 11 and 12). The largest central hole was programmed by a spiral compiler built-in function (line 18, Figure 13). A set holes to build the cross were programed in line 48 of Figure 13 with the function **holesLine** by defining the center coordinates of the first one (`(X1ini,Y1ini)`), the depth of the holes (`DepthBC`) and the upper retract plane (`Zseg`). Finally, the holes in circumference pattern were programed with the built-in function **circArray** in line 56 of the Figure 13. This function is able to create any number of arc-equidistant holes placed in a circumference of any diameter

International Science Index, Computer and Information Engineering Vol:6, No:7, 2012 waset.org/Publication/5580

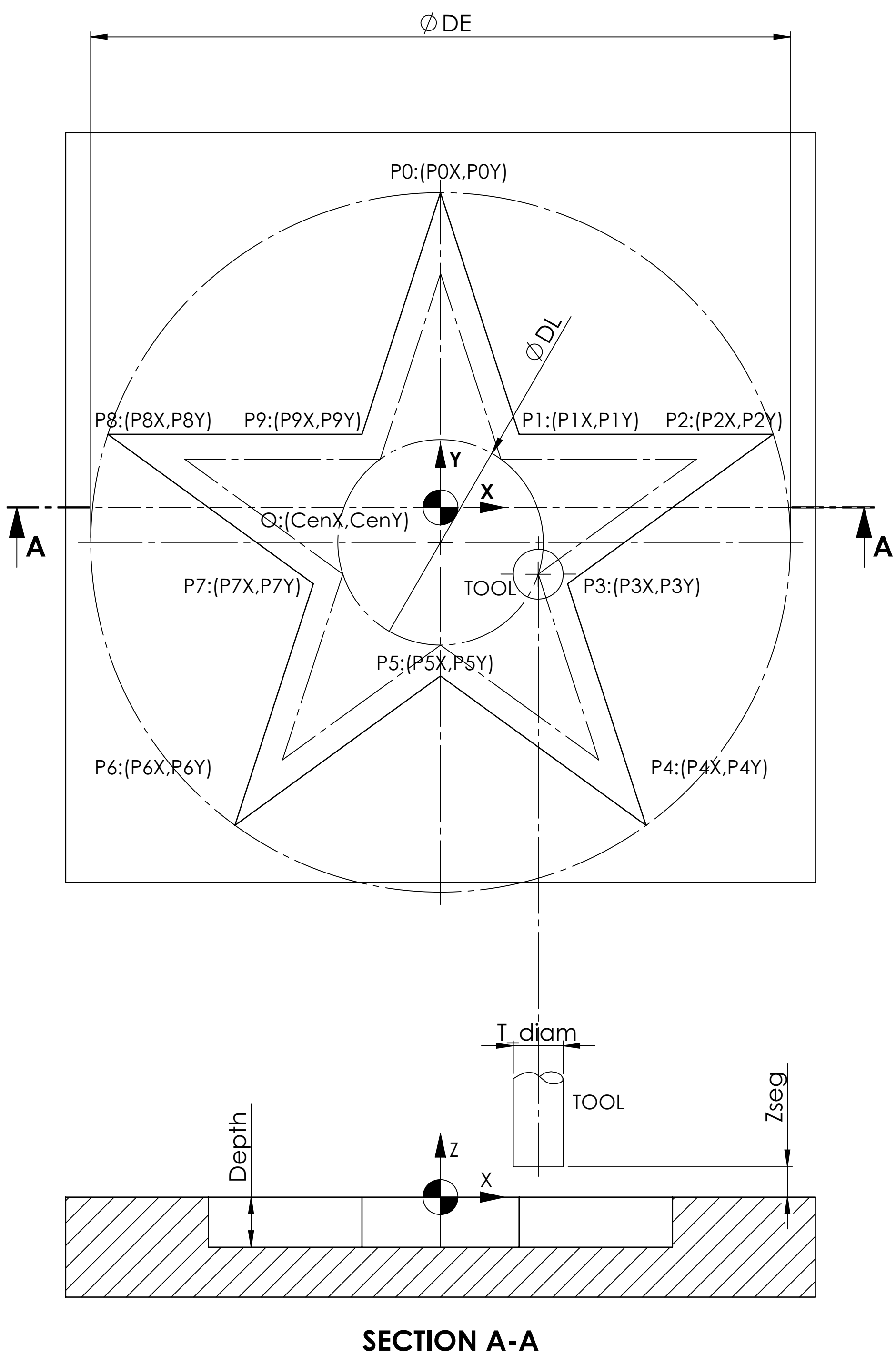

Fig. 8. Geometric parameters of the designed Star Pocket part.

International Science Index, Computer and Information Engineering Vol:6, No:7, 2012 waset.org/Publication/5580

```
 1: O002;       Star Pocket Part Program
    ...
 6: DE = 140
 7: CenX = 0
 8: CenY = −7
    ...
23: while (DPass <= abs(Depth))
24: {
25:    while (DL ∗ OV_LAP > T_DIAM)
26:    {
27:       P0X = CenX
28:       P0Y = CenY + DM/2
          ...
47:       G0 X<P0X> Y<P0Y> Z<Zseg>
48:       G1 Z<-DPass>
49:       X<P1X> Y<P1Y>
          ...
62:    }
       ...
69: }
    ...
72: M30
```

Fig. 9. EGCL program for producing the Star Pocket in Figure 7.

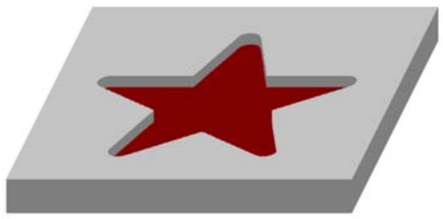

Fig. 10. Star pocket part simulated in CncSimulator© software ([14]).

by specifying the number of holes (`NumElem`), circumference radius (`Radius`) and center (`(Cenx,Ceny)`), the depth of the holes (`DepthBC`) and the upper retract plane (`Zseg`). This case study illustrates the built-in function advantages of EGCL for programming geometric patterns of machining features.

The afore mentioned application cases (sections IV-A and IV-B), would require long programming time programmed manually. To illustrate this contrast, input and output files were compared (Table I). Additionally, programs manually written are valid only for the set of coordinate points for which they were created, while macro-based programming on EGCL can

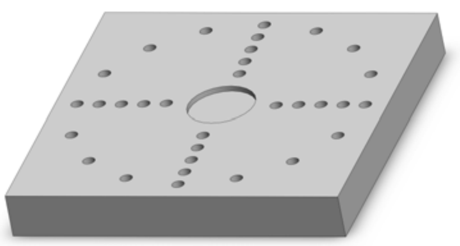

Fig. 11. Proposed Drill Target Pattern part.

```
 1: O003;
    ...
 6: CenX = 0
 7: CenY = 0
 8: Depth = −2.5
 9: Zseg = 5
    ...
18: spiral(Pitch, Rext, CenX, CenY, Depth, Zseg)
    ...
26: NumHoles = 4
27: dx1 = 10
28: dy1 = 0
29: Xini1 = 25
30: Yini1 = 0
31: DepthBC = −5
    ...
48: holesLine(NumHoles, dx1, dy1, Xini1, Yini1, DepthBC, Zseg)
    ...
52: NumElem = 16
53: Radius = 65
54: Cenx = 0
55: Ceny = 0
56: circArray(NumElem, Radius, Cenx, Ceny, DepthBC, Zseg)
    ...
67: M30
```

Fig. 13. EGCL program for producing the Drill Target Pattern part in Figure 11.

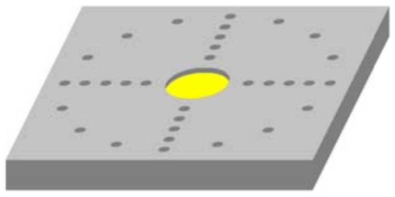

Fig. 14. Drill Target Pattern part simulated in CncSimulator© software ([14]).

re-use code by simply modifying the mnemonic geometric variables defined at the header of the programs.

V. Conclusion and Future Work

This article presents the implementations of a language extension for G-code (EGCL). The compiler for ECGL produces ISO compliant elementary G-code as output. The output programs of the compiler are portable due to the fact that the most CNC machines are able to process this basic ISO G-code. The developed EGCL has the following characteristics: (1) it allows to use of mnemonic variable names, (2) it permits symbolic coordinates, (3) it accepts flow control structures, such as IF and WHILE, and (4) it allows to use built-in geometric functions (e.g. 2D spiral, geometric holes arrays). These capabilities save time in part programming and make re-programmation easier and more immune to errors.

	Star Pocket	Drill Target Pattern
Lines in the EGCL file	72	64
Lines in the IGCL file	257	236

TABLE I
Comparison of number of lines of EGCL vs. IGCL equivalent programs.

International Science Index, Computer and Information Engineering Vol:6, No:7, 2012 waset.org/Publication/5580

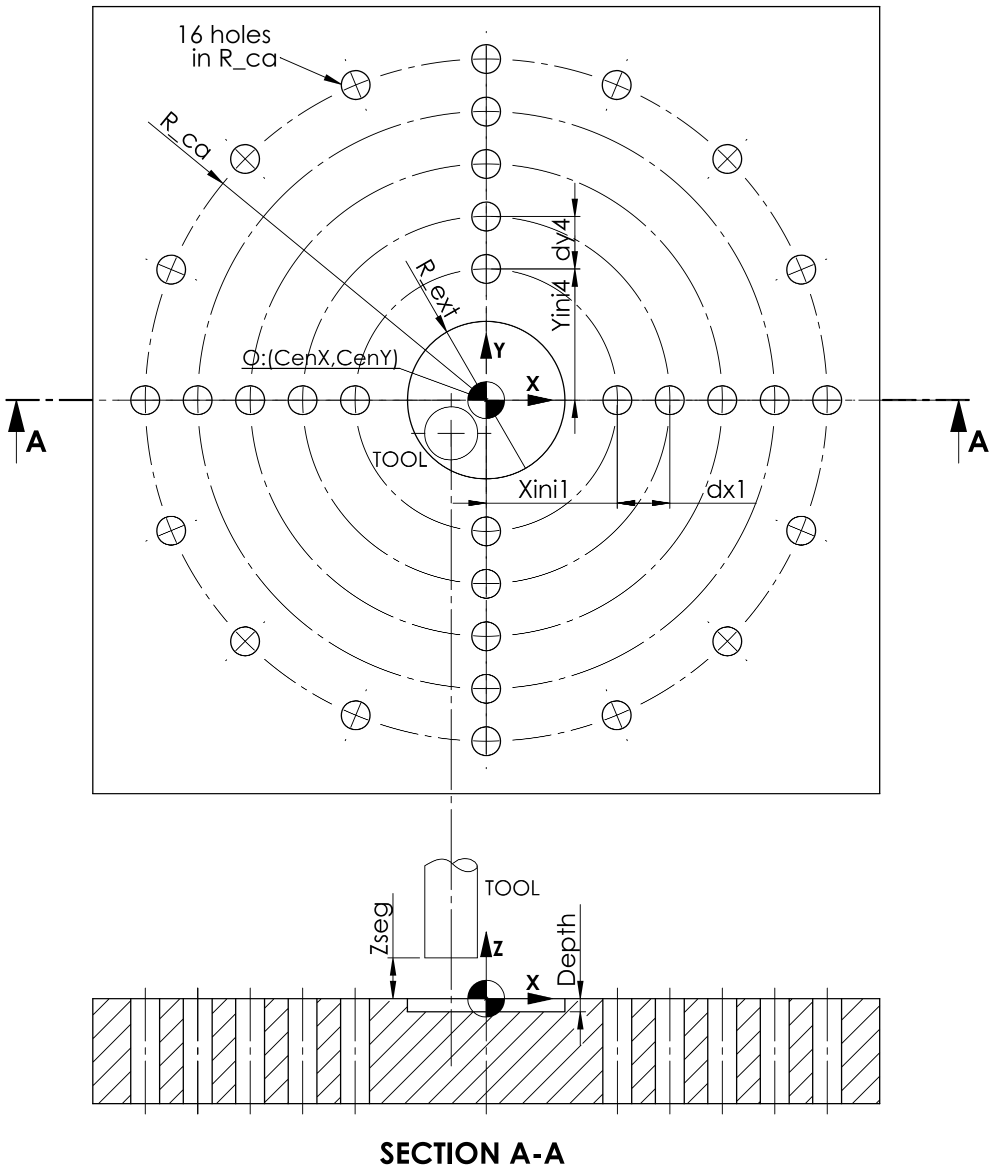

Fig. 12. Geometric parameters of the designed Drill Target Pattern part.

The EGCL grammar was achieved by extending IGCL with arithmetic and boolean expressions, flow control statements (e.g. IF, WHILE and function use capabilities). The lexical, syntactic and intermediate G-code generation software were accordingly implemented. The code generator was created to produce the output file in IGCL.

Future work includes allowing the part-programmer to predefine own geometrical functions in both, source and object form and to use them as libraries for the EGCL programs. Likewise, we seek to optimize the code (following criteria usual in CNC machining) and conduct geometrical verifications by using the middle-end of the compiler.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006.

[2] G. Arroyo, C. Ochoa, J. Silva, and G. Vidal. Towards CNC Programming Using Haskell. *Advances in Artificial Intelligence–IBERAMIA 2004*, pages 386–396, 2004.

[3] X. Guo, Y. Liu, D. Du, K. Yamazaki, and M. Fujishima. A Universal NC Program Processor Design and Prototype Implementation for CNC Systems. *The International Journal of Advanced Manufacturing Technology*, pages 1–15, 2011.

[4] S Habeeb and X Xu. A Novel CNC System for Turning Operations Based on a High-Level Data Model. *The International Journal of Advanced Manufacturing Technology*, 43:323–336, 2009.

[5] International Organization for Standardization (1982). ISO 6983-1: Numerical control of machines – Program format and definition of address words – Part 1: Data format for positioning, line motion and contouring control systems. International Organization for Standardization, Geneva, Switzerland.
[6] International Organization for Standardization (2004). ISO 14649-1: Industrial automation systems and integration – Data model for computerized numerical controllers – Part 1: overview and fundamental principles. International Organization for Standardization, Geneva, Switzerland.
[7] SC. Johnson. YACC - Yet Another Compiler-Compiler. *Computer Science*, Technical Report 32, 1975.
[8] ME Lesk. Lex - A Lexical Analyzer Generator. *Computing Science*, Technical Report 39, 1975.
[9] Y. Liu, X. Guo, W. Li, K. Yamazaki, K. Kashihara, and M. Fujishima. An Intelligent NC Program Processor for CNC System of Machine Tool. *Robotics and Computer-Integrated Manufacturing*, 23(2):160–169, 2007.
[10] S. Nagle and J. Wiegley. GSP: Extending G-Code Using JSP Servlet Technologies. In *Automation Science and Engineering, 2008. CASE 2008. IEEE International Conference on*, pages 953–958. IEEE, 2008.
[11] D.T. Ross. The Design and Use of the APT Language for Automatic Programming of Numerically Controlled Machine Tools. In *Proc. 1959 Computer Applications Symposium, Chiergo*, pages 80–99, 1959.
[12] T. Schroeder and M. Hoffmann. Flexible Automatic Converting of NC Programs. A Cross-compiler for Structured Text. *International Journal of Production Research*, 44(13):2671–2679, 2006.
[13] S.J. Shin, S.H. Suh, and I. Stroud. Reincarnation of G-code Based Part Programs Into STEP-NC for Turning Applications. *Computer-Aided Design*, 39(1):1–16, 2007.
[14] Bulldog Digital Technologies. Free CNC-Simulator (Version 4.53f) [Software]. Avaliable from http://www.cncsimulator.com, 2007.
[15] J. Zhang, L. Wang, S. Li, and F. Zhang. The Research of an Intelligent Open CNC System. *Initiatives of Precision Engineering at the Beginning of a Millennium*, pages 779–783, 2002.

International Science Index, Computer and Information Engineering Vol:6, No:7, 2012 waset.org/Publication/5580