ATL – A Teaching Language

Philip A. Nelson

1 Introduction

ATL first started around 1980 for a Compiler Construction class taught at Pacific Union College in Angwin, California. It has changed over the years while being used for Compiler classes at Western Washington University. This language is a block structured language based on the Algol family of languages. It has several features that are in specific response to issues in other languages in the Algol family. For example, the assignment operator is a `<--` symbol so as to not be confused in any way with any kind of equality operator. Also, IFs and other similar constructs end similar to "end if". (e.g. "while ... do .... end while".)

For use in Compiler Construction classes, we have defined subsets of ATL to make building specific projects easier to understand. For example, ATL/0 is a very small language that has no user defined procedures and only simple computation with integers. ATL/1 is a subset that has been used successfully for a quarter project for Compiler Construction classes. ATL/2 contains constructs that make the language a more featureful language.

The remainder of this document describes each of the subsets of ATL in details so that it may be implemented.

2 ATL/0

ATL/0 is designed to be a very small language so that a recursive descent compiler can be written without any formal tools for lexical analysis or parsing. It also needs to be small enough to be able to write this compiler in about two weeks. As such it contains only basic constructs. All ATL/0 programs follow the following outline: (Items that look like <words> are descriptions of what goes at that location.)

```
program <program_name>;

  variable <namelist> : integer ;

  begin
    <statement list: read, write, writeln, or assignment>
  end <program_name>.
```

2.1 Lexical Tokens

The lexical tokens for ATL/0 are:

- *Reserved Words:* end, read, begin, write, integer, program, writeln, and variable. These reserved words are composed of only lower case letters.
- *Integer Constants:* A sequence of one or more digits (0-9).
- *Identifiers:* A letter followed by any number of letters, digits or underscore characters. All letters are lower case letters.
- *String:* A string is composed of a double quote followed by characters until the second double quote. There is no way to include a double quote in the string.
- *Assignment operator:* The characters `<--`
- *Other single character tokens:* + - ( ) ; , : .

2.2 Grammar

In this grammar, **bold face** words are reserved words, *id* is an identifier and non-bold face words are variables in the grammar, **STRING** is a string token, **INTCONST** is an integer constant and quoted characters are other tokens. Constructs enclosed by braces may be repeated zero or more times.

goal → **program** *id* ";" var_decl **begin** statements **end** *id* "."
var_decl → **variable** id_list ":" **integer** ";"
id_list → *id* { "," *id* }
statements → statement ";" { statement ";" }
statement → *id* "<--" expr
| **read** "(" read_list ")"
| **write** "(" write_list ")"
| **writeln** "(" ")"
read_list → *id* { ";" *id* }
write_list → write_elem { ";" write_elem }
write_elem → **STRING**
| expr
expr → primary { add_op primary }
primary → "(" expr ")"
| *id*
| **INTCONST**
add_op → "+"
| "-"

In years when CS520 is taught concurrently, the following change and addition to rules for ATL/0 is done to give the CS520 a slightly harder problem. This adds the reserved word "mod".

Replace the "expr" rule with:

expr → factor { add_op factor }

Add the following rules:

factor → primary { mul_op primary }

mul_op → "*"
| "/"
| **mod**

2.3 Semantics

The semantics of ATL/0 are quite simple. First, the program name and the name at the end of the program must match. Variables must be declared before use and may not be the same name as the program. "write" statements write exactly that string or that integer with no spaces. "writeln" causes the output to go to a new line. Assignment statements compute integer expressions as defined by the grammar.

3 ATL/1

ATL/1 adds a variety of features to make it a reasonable project for a quarter long class on Compiler Construction. Primary additions are functions and procedures with value and variable parameters and the if, while and repeat control structures. Functions and procedures may be nested as in Pascal. More basic types and arrays are added to the type system. Along with the additions, the ATL/0 keywords "integer", "read", "write", and "writeln" became just pre-declared identifiers in ATL/1.

3.1 Lexical Tokens and Scanner Issues

This section details the issues for the lexical analysis phase of an ATL/1 compiler.

- *Comments:* Comments are started with "(*" and are terminated at the *matching* "*)". This allows for nested comments.
- *Reserved Words:* In ATL/1 all capitalizations of a reserved word is the same reserved word. This allows the user to choose their favorite capitalization of reserved words. The words are: do, if, is, of, or, and, end, not, else, then, type, array, begin, elsif, until, value, while, repeat, return, returns, program, variable, function, and procedure.
- *Integer Constants:* A sequence of one or more digits (0-9).

- *Identifiers:* A letter followed by any number of letters, digits or underscore characters. Upper case letters are included.
- *String:* A string is composed of a double quote followed by characters until the second double quote. Strings may not cross line breaks. Strings may have "quoted" characters in the string. They are `\b` for backspace, `\f` for form-feed, `\n` for newline, `\r` for carriage return, `\t` for tab, `\"` is the double quote character and `\\` for the backslash character. An arbitrary character can be specified by `\nnn` notation where nnn is a decimal value less than 256.
- *Assignment operator:* The characters `<--` and denoted ASSIGN in the grammar below.
- *Other character tokens:* `+ - * / ( ) ; , : . = != < <= > >=`

3.2 Grammar

In this grammar, upper case words are reserved words or other lexical tokens. Lower case words are variables. Also, REL_OP means one of `= != < <= > >=` and MUL_OP means one of * / mod, where mod is a reserved word.

program → PROGRAM IDENTIFIER ';' block '.'
block → declaration_list BEGIN statement_list END IDENTIFIER
declaration_list → λ
| declaration_list declaration
declaration → type_decl
| var_decl
| subprogram
type_decl → TYPE type_dec_list
type_dec_list → type_dec
| type_dec_list type_dec
type_dec → IDENTIFIER IS type ';'
var_decl → VARIABLE var_dec_list
var_dec_list → id_list ':' type ';'
| var_dec_list id_list ':' type ';'
id_list → IDENTIFIER
| id_list ',' IDENTIFIER
type → type_identifier
| ARRAY '[' const ':' const ']' OF simple_type
/* For a scalar type = INTEGER or BOOLEAN. */ simple_type → type_identifier
type_identifier → IDENTIFIER

subprogram → PROCEDURE IDENTIFIER '(' fparam_list ')' ';' block ';'
| FUNCTION IDENTIFIER '(' fparam_list ')' RETURNS simple_type ';' block ';'
fparam_list → λ
| fp_list
fp_list → fparam
| fp_list ';' fparam
fparam → VALUE id_list ':' IDENTIFIER default
| VARIABLE id_list ':' IDENTIFIER
default → λ
| ASSIGN const
| ASSIGN IDENTIFIER
statement_list → λ
| statement_list statement ';'
statement → variable ASSIGN expression
| IF expression THEN statement_list elsif_clause else_clause END IF
| REPEAT statement_list UNTIL expression
| WHILE expression DO statement_list END WHILE
| IDENTIFIER '(' maparam_list ')'
| RETURN
| RETURN expression
elsif_clause → λ
| elsif_clause ELSIF expression THEN statement_list
maparam_list → aparam_list
| maparam_list ';' aparam_list
aparam_list → aparam
| aparam_list ',' aparam
aparam → λ
| expression
expression → expression '-' expression
| expression '+' expression
| expression MUL_OP expression
| expression AND expression
| expression OR expression
| expression REL_OP expression
| NOT expression
| '-' expression
| '(' expression ')'
| IDENTIFIER '(' aparam_list ')'
| variable
| STRING

| INT_CONST
const → INT_CONST
| '-' INT_CONST
variable → IDENTIFIER
| IDENTIFIER '[' expression ']'

3.3 Semantics

The semantics of ATL/1 are very similar to other Algol family languages like Pascal or Modula-2 that include procedure and function declarations inside other procedures or functions. The following is a list of semantic issues:

- Data types: ATL/1 has three kinds of simple data, booleans, integers and string constants. (The addition of the real data type adds complexity to the compiler, but not a lot. For this reason, reals are not a part of the standard data types for ATL/1.) String constants are limited to one use in ATL/1, that is passing them to the write() procedure.
- Structured data types: ATL/1 has only single dimensional arrays that are indexed by integers.
- Assignment: The assignment operator assigns simple data and arrays. No data conversion is done by the assignment operator.
- Evaluation order: If a subscripted array is on the left hand side of an assignment statement, the subscript expression is evaluated before the right hand expression is evaluated.
- Operator presidence: highest to lowest, all binary operators are left associative.
  - unary minus
  - MUL_OP (* / mod)
  - + -
  - REL_OP (= ! = < <= > >=)
  - NOT
  - AND
  - OR
- Boolean expressions must not be evaluated with short circuit evaluation.

- Type system: ATL/1 is a strongly typed language. Binary operations require the same type of data for both operations. Relation operations operate on both boolean and integer types and the result type is boolean. The mathematical operators operate only on integers. None of the mathematical or relational operators operate on arrays.

- Scope: The scope of a definition is from the definition to the end of the enclosing procedure or function. The scope of definitions in the main program are to the end of the main program and are global variables to the program. Nested procedures and functions may use variables that are in scope, whether global or in a procedure or function that encloses the use of the variable.

- Parameters: Parameters are either pass by variable or pass by value. Pass by value parameters are "read-only" inside the called procedure. They may not appear on the left hand side of assignment. They may not be used as an actual parameter for a call by variable parameter. This eliminates the common beginner's logic error of assigning an expression to a call by value parameter and expecting changes outside of the procedure. The final thing to note is the the programmer must specify what kind of parameter they want for every parameter or list of parameters. A typical procedure head looks like:

  Procedure TestProc (Value x : integer; Variable y : boolean);

- Default parameters: ATL/1 supports default values for call by value parameters in functions and procedures. The default value must be a constant, a named constant or the name of a in-scope variable. Any value parameter may be given an default value. The call to a procedure with default parameters may simply not provide an actual parameter for the corresponding argument. Consider the procedure:

  Procedure proc1 (Variable a: integer; Value b: integer <-- 35;
  Value c : boolean <-- true); ...

  The call "proc1(x,,y)" is legal (assuming correct types of x and y) since the second argument (b) has a default value. The form "proc1(x)" is legal since both b and c have default values. Any value parameter may have a default value and subsequent parameters may be either call by value or call by variable. Subsequent value parameters do not have to have default values.

- Procedure Overloading: ATL/1 supports procedure and function overloading. Overloading is done only on the basis of the types of the non-default arguments. The kind of parameter, variable or value, is not used to determine unique parameter lists for overloading. Procedures and functions of the same name may have identical parameter lists because their calling sequence can uniquely determine if a procedure or a functions is being called.

- Special procedure calls: ATL/1 supports a special syntactic sugar for repeatedly calling procedures with the same name over and over again. The code "proc(x; y; z);" is identical to seeing the code "proc(x); proc(y); proc(z);". This work with any number of arguments. For example, "proc (1,2,3; a; 3-5,10);" is the same as "proc(1,2,3); proc(a); proc(3-5,10);". This makes sense only if proc is overloaded and each call to proc may call a different procedure. This syntax was added to make I/O much easier to construct using a single named procedure for writing data: "write(1; true; "string");"

- Function and Procedure Returns: Functions must execute a "return expr;" statement. Running off the end of the function produces a run time error. Procedures may either execute a "return;" statement or run off the end of the procedure which then does the same as a return statement.

- Naming considerations: The name of the procedure must match the name given after the "end" of the procedure body. The program name must match the name at the end of the program. No other global name may be the same as the program name.

- Predefined names: ATL/1 has several names that are predefined. These can be considered a "super global" namespace. (In ATL/2, these names are in a "std" module.) The names and definitions of them are:

  - integer – a simple type name
  - boolean – a simple type name
  - true – constant of type boolean
  - false – constant of type boolean
  - procedure read (variable s : integer) – reads an integer, leading blank space and end-of-lines are ignored.
  - procedure readln ( ) – throws away the current input line.
  - procedure write (variable str : string; value field : integer <-- 0) – writes a string, field size of 0 writes only the string. If field size is smaller than the string, only the first field size characters are written. If field size is larger than the string size, leading blanks are printed to pad the string to write exactly field characters. Note: ATL/1 is implemented so a string constant matches the first argument and is placed in temporary storage so it may be passed to write(). This is the only place where type "string" is allowed in ATL/1.
  - procedure write (value expr : integer; value field : integer <-- 0) – writes an integer. This will always write the value of the integer regardless of the value of field. If field is larger than the number of digits needed to print the integer, leading spaces are used so write writes exactly field characters.

- procedure write (value expr : boolean; value field : integer <-- 0) – prints the word “true” or “false”. The field argument modifies this the same way as it does for strings.
- procedure writeln () – write a newline
- function abs (value x : integer) – returns the absolute value of x.

4 ATL/2

ATL/2 is the most fluid of all these languages. It originally was specified for use in a graduate level compiler class. In 2008 it was redefined to the current definition and associated implementation as a real introductory programming language that could help students understand programming in a much safer environment than currently is used by most schools. (Note: Many people disagree with using a language like ATL/2 as an introductory language. In other pursuits we do not put new learners in a professional environment. New pilots train in small, simple to use airplanes. They do not start learning to fly on a 747 jet. New drivers do not learn to drive in any kind of a race car. In computer science we expect new programmers to learn in the full professional version of the tool, java, C++, python and so forth.)

The following features have been added on top of ATL/1:

- Real and character basic types and constants of those types
- Enumerated data types
- Records
- Multi-dimensional arrays
- For statements
- A simple module system and separate compilation
- Restricted pointers and dynamic memory allocation
- Open Array Parameters used to pass arbitrary sized arrays
- Initialization section added following variable declarations, including array and record initialization
- A standard module that defines the standard type names and a collection of standard types, procedures and functions.

4.1 Lexical Tokens and Scanner Issues

With the additions, there are new tokens that for ATL/2. The additional items are:

- New Reserved Words: constant delete export import initialize module new pointer
- Character constants: 'x' or '\c' where c is one of b, f, n, r, or t.
- Real constants: D*.D+[eD+], it requires the decimal point, but leading digits are optional. D is a single digit (0-9). The "e" may be lower or upper case and if an "e" exists, there must be at least one digit following it.
- The "follow the pointer" operator: `->`

4.2 Grammar

compile_unit → PROGRAM IDENTIFIER ';' import_list block '.'
| MODULE IDENTIFIER ';' import_list declaration_list END IDENTIFIER '.'
import_list → IMPORT id_list ';'
block → declaration_list BEGIN statement_list END IDENTIFIER
declaration_list → λ
| declaration_list export declaration
export → λ
| EXPORT
declaration → type_decl
| var_decl
| const_decl
| subprogram
type_decl → TYPE type_dec_list
type_dec_list → type_dec
| type_dec_list type_dec
type_dec → IDENTIFIER IS type ';'
var_decl → VARIABLE var_dec_list initialize
var_dec_list → id_list ':' type ';'
| var_dec_list id_list ':' type ';'
initialize → λ
| INITIALIZE initialize_list
initialize_list → init_element
| initialize_list init_element
init_element → IDENTIFIER ASSIGN init_value ';'

init_value → expression
| '[' value_list ']'
| '{' value_list '}'
value_list → init_value
| init_list ',' init_value
const_decl → CONSTANT const_dec_list
const_dec_list → const_dec
| const_dec_list const_dec
const_dec → IDENTIFIER IS expression ';'
id_list → IDENTIFIER
| id_list ',' IDENTIFIER
type → type_identifier
| '(' id_list ')'
| ARRAY '[' expression ':' expression ']' OF type
| RECORD field_list END RECORD
| POINTER TO type_identifier
field_list → id_list ':' type ';'
| field_list id_list ':' type ';'
type_identifier → qualified_identifier
subprogram → PROCEDURE IDENTIFIER '(' fparam_list ')' ';' block ';
| FUNCTION IDENTIFIER '(' fparam_list ')' RETURNS type ';' block ';'
fparam_list → λ
| fp_list
fp_list → fparam
| fp_list ';' fparam
fparam → VALUE id_list ':' IDENTIFIER default
| VARIABLE id_list ':' IDENTIFIER
default → λ
| ASSIGN expression
| ASSIGN IDENTIFIER
statement_list → λ
| statement_list statement ';'
statement → variable ASSIGN expression
| IF expression THEN statement_list elsif_clause else_clause END IF
| REPEAT statement_list UNTIL expression
| WHILE expression DO statement_list END WHILE
| qualified_identifier '(' maparam_list ')'
| RETURN
| RETURN expression
| FOR IDENTIFIER ASSIGN expression TO expression by_clause DO statement_list

END FOR
    | NEW variable
    | DELETE variable
elsif_clause → λ
    | elsif_clause ELSIF expression THEN statement_list
by_clause → λ
    | BY expression
maparam_list → aparam_list
    | maparam_list ';' aparam_list
aparam_list → aparam
    | aparam_list ',' aparam
aparam → λ
    | expression
expression → expression '-' expression
    | expression '+' expression
    | expression MUL_OP expression
    | expression AND expression
    | expression OR expression
    | expression REL_OP expression
    | NOT expression
    | '-' expression
    | '(' expression ')'
    | qualified_identifier '(' aparam_list ')'
    | variable
    | STRING
    | INT_CONST
    | REAL_CONST
    | CHAR_CONST
variable → qualified_identifier
    | variable '[' expression ']'
    | variable '.' IDENTIFIER
    | variable FOLLOW
qualified_identifier → IDENTIFIER
    | IDENTIFIER '!' IDENTIFIER

Note: While not specified in this grammar, the current implementation of atl/2 that targets "hc" includes a way to specify a "built-in" procedure or function. The rules in the implementation grammar are additions to the "subprogram" rule as follows:

subprogram → BUILTIN PROCEDURE IDENTIFIER '(' fparam_list ')' ';' STRING ';
| BUILTIN FUNCTION IDENTIFIER '(' fparam_list ')' RETURNS type ';' STRING ';'

The string instead of the body is a 'hc' instruction that is used to call the function or procedure. The builtin feature is enabled only for compiling the standard module "std.atl". It is not available on the regular compiler.

4.3 Semantics

- FOR statement: The control variable is a new variable who's scope is the FOR statement. The type of the control variable is the same as the type of the initial expression. The final expression must be the same type as the initial expression and BY expression must be an integer. All types must be integral types, enumerated, characters, or integers. If the name of the control variable conflicts with another name, it hides the other name. The control variable is a read only variable in the body of the loop.
- Modules: This needs documentation.

Started: December 2011
Updated: September 2018