Testing Student-Made Compilers

José de Oliveira Guimarães
Departamento de Computação
UFSCar - São Carlos
e-mail: jose@pink.ifqsc.usp.br
Brasil

24 de outubro de 2007

Resumo

This article presents a few guidelines and examples for testing Pascal compilers designed by students in introductory courses. A standard Pascal subset with little alteration is used, and the possible sources of errors are divided into eight categories. Examples and probable sources of error are exposed for each category.

1 Introduction

Program tests have been neglected in computer science courses, particularly in introductory compilation courses. Correctness in compilers is often hard to be obtained for several reasons:

- They make use of complex data structures where pointers and dynamic allocation of memory are widely used.
- They must resemble the syntax and semantics of a particular language whose definition is often innacurate. A comprehensive knowlege of the language semantics is required.
- The number of possible constructions and construction combinations is extremely high. Sometimes it is necessary to generate a different code if a particular characteristic or construction is in presence of another. For instance, upon calling the subroutine
  `P(x)`
  the real parameter (`x`) may be taken as local variable, formal parameter by value, or reference. Also the cases in which the formal parameter of `P` is by value and reference should be considered.

The points discussed in this article relate to a simplified Pascal [2] compiler. The information herewith presented can be used in building tests for compilers designed in introductory courses. The article is organized as follows: Section 2 is a preliminary discussion on software-testing methodologies. Guidelines on building tests for a Pascal compiler are presented in section 3. Section 4 shows the conclusion of the article.

2 Program-Testing Methodologies

The methodologies for testing programs can be roughly classified as human or non-human. The main human methodologies are inspections and walkthrough [1]. They are carried out by a group of people that analyse the program under test. In a walkthrough, the program is manually executed for a few test situations, while in an inspection its logical conception is described by the programmer as the other participants investigate its validity. The program will be checked with the help of a list of frequent errors, e.g. non-initialized variables.

The second type of methodologies (non-human) involves running the program to be tested with a subset of data extracted from the input dominion, according to the testing criteria. Such criteria generally fall within one of the following test techniques:

- Black box tests. The required test elements are defined without using the source code of the program under test. Only the description of its input is employed. Ideally, there should be a test for each possible program input, which is not a feasible task. An example of a black box test is the analysis of boundary values. This method tries to create inputs that fall near the limits estimated by the program specifications.

- White box tests. In this case, the elements required for the test are identified from the knowledge of a particular implementation structure. Ideally, there should be a test case for every possible execution flow admitted by the program, which is again not viable in practice.

The suggestions presented in this article are black box tests, since they presume the testing of a generic Pascal compiler whose source code is not available.

3 Designing the Set of Test Cases

The guidelines for detecting compiler errors presented in this article were made to be used in the subject "Compiler Design II", at the Federal University of São Carlos, Brazil. They can be divided into eight categories:

- Insufficient knowledge of Pascal language
- Boundaries and unusual situations
- Errors in analysing literal integer constants
- Incorrect code generation
- Errors with the symbol table
- Errors in the lexic analyser
- Errors in the syntax analyser
- Errors in the semantic analyser

Each category has a set of remarks and examples that can be employed in building tests for the compilers. Examples that could fit in more than one category were arbitrarily placed in one of them.

The Pascal subset used excludes:

- Commands `for` and `repeat until`.
- Sets, scalar types, declaration of constants.
- Records.

The only data types of data are `integer` and `boolean`. Array indexes cannot be of the `boolean` type. Arrays can be passed as parameters, returned from functions and used in attribution operations.

The code generated by the student-made compilers is a pseudo-code for a hypothetic machine. The instructions on this machine manipulate adequately the problems originated by the block structure of Pascal.

The guidelines are shown more detailed below.

3.1 Insufficient Knowledge of Pascal Language

The semantic details listed below are permitted/prohibited in the language but often ignored by the students:

- The words `read`, `write`, `boolean`, `integer`, `true`, and `false` are not reserved words and thus can be used as identifiers.
- The type of a subroutine parameter must be basic (`integer` or `boolean`), or declared with type. An illegal example could be:

```
procedure P(a: array[1..10]
               of integer);
```

- The elements of a array can be of any type, including another array. For instance:

```
var a: array[1..10] of array[1..10]
       of integer;
```

- Comparison operations (`>` `>=` `<` `<=`) can be performed upon `boolean` type values.
- The predefined procedure `write` can write integers only. Writing `boolean` values is not permitted in the Pascal dialect used. Similarly, the `read` procedure does not take `boolean` parameters.
- A subroutine may have up to 32 parameters in the Pascal dialect used.

3.2 Limits and Unusual Situations

This subsection presents examples and situations that test the capacity of a compiler to manipulate borderline, pathologic, and machine-dependent situations. Some of these situations are:

- The Pascal program to be compiled has:

  [1] 0 bytes

  [2] 64 Kbytes (65536 bytes)

  [3] 64 Kbytes - 1 (65535 bytes)

  [4] 64 Kbytes - 2 (65534 bytes)

  [5] 32 Kbytes (32768 bytes)

  [6] 32 Kbytes - 1 (32767 bytes)

  [7] 32 Kbytes - 2 (32766 bytes)

  [8] 64 Kbytes and containing only the SPACE character (ASCII 32).

  Some of the files above may lead to compiler errors due to the limit values that an integer can store:

  [1] -32,768 to 32,767 for a two-byte integer, with signal.

  [2] 0 to 65,535 for a two-byte integer, without signal.

  An error example is: integer `N` with signal contains the file size, which is 32,767 bytes. In case the expression `N+1` is used, an error will occur. `N+1` is equal to -32,768, which is not what the programmer expected.

- The hard disk or floppy disk does not have enough space to write the code generated by the compiler (assuming that the code will be written to disk).

- Memory runs out during dynamic memory allocation (with Pascal's `new` or C's standard library `malloc`) by the compiler. In this case the simplest thing to do is terminate the compilation. A less drastic solution would involve the return of an error condition by the routine where the memory overflow took place. The routine from which it was called would also end its running and return an error signal, and so on. Two possible errors with the compiler (or any program) are:

  [1] The success of memory allocation is not verified.

  [2] The following situation occurs: procedure `P` calls (at run time) routine `Q` which calls routine `R` which in turn calls routine `S`. A memory allocation failure happens in the latter, leading `S` to terminate the execution and return an error signal in one of its parameters. Upon calling `S`, `R` verifies, by checking the parameter, that an error has occurred in `S` and terminates its execution, passing on the error signal to `Q` which ignores the parameter of `R` and signals with the error:

  ```
  /* In language C, within Q */
  R( &Error );

  /* Variable &Error is no longer
     used in Q */
  ```

- A recursive descendent analyser makes use of recursiveness to analyse syntax. The compilation of a program with subroutine nestings (one within the other) involves recursive calls on the compiler subroutines. If too many nestings exist, there will be too many recursive calls, which may overload the computer stack memory. This is limited to 64 Kb in the IBM PC. If the compiler has been compiled with too small a stack size, memory may run out even during the compilation of programs with few nestings.

3.3 Errors in the Analysis of Integer Constants

The language subset defined for the compiler does not accept real numbers. Nevertheless, there can be a number of errors in the analysis of integers. According to the Pascal specification used, integer values must lie between 0 and 32,767. Negative numbers will be obtained from the least unary operator. Thus, -32,678 is an illegal number. A few details to be considered are:

- Numbers 32,769 and 32,768 are illegal, whereas 32,767 and 32,766 are legal.

- A number must not be followed by a letter. If the analyser were scanning letters and numbers up to five characters (the largest possible number, 32,767, has five characters), an error would probably be signalled in the example shown in Figure 1. This might not happen with the example of Figure 2.

```
procedure P;
  var a : integer;
begin
a:= 12end;
```

Figura 1: Keyword at the end of a number

```
procedure P;
  var a : integer;
begin
a:= 12345end;
```

Figura 2: Keyword at the end of a 5-character number

- A number can have any number of zeros preceding the first non- zero digit, if the latter exists. The following numbers are therefore valid:

  ```
  0000000000000000000000       {number 1}
  0000000000000000012345       {number 2}
  ```

  These numbers may lead the analyser to comitting several mistakes:

  - The maximum number of digits of the greatest integer (32,767) is five. Based on this, the analyser may consider (number 1) and (number 2) as outrangers.
  - If the analyser skips all zeros preceding a non-zero digit, it may signal an error from not finding a non-zero digit in (number 1). Or it might accept the number, assigning it another value like the algorithm below:

    ```
    /* N not initialized */
    if character  = '0'
    then
      while  character = '0' do
        get next character
    endif
    if character is digit
    then
      N = 0;
      while  character is digit do
         N = 10*N + ord(character)
             - ord('0')
    endif
    /* with (number 1), n will hold an
       undefined value at this point of
       the program */
    ```

```
/*  language C */
n = 0;
i = 0;
while ( *s != '\0' && i < 6 ) {
  n = 10*n + *s - '0';
  s++;
  i++;
  }
if ( n >= 32768  ||  i == 0 ) ERROR();
```

Figura 3: Example of an incorrect code for the convertion of a string into a number

- The programmer has the option, as numbers are scanned, of placing its characters in a string:

  ```
  S = '';      /* empty string */
  while character is digit do
     append ch to the end of S
     get next character
  endwhile
  ```

  If there are not enough string positions for a given number, and if there is no checking upon the maximum number of characters the string can store, there will be use of non-allocated memory for this variable, which brings about a compiler crash.

- An error in the conversion routine of characters to numbers may lead the compiler to accepting values greater than 32,767. An example of faulty coding is given in Figure 3. In the IBM-PC, if `n` is considered as an unsigned integer (unsigned int in C, maximum value = 65,535), no error would be signaled upon analysing 70,000. It would be considered as 70,000-65,536, hence less than 32,768.

3.4 Incorrect Code Generation

Sometimes the code for operators, commands, or language instructions is incorrectly generated. Correcting this type of error (in simple, non-optimized codes) is often uncomplicated. A few remarks on code generation errors are given below:

- The compiler may generate a code for a certain operator in place of another (e.g. `>` instead of `>=`).

- Some machine instructions for which the code is generated comprise parameters such as the current lexic level (subroutine nesting level), the number of words occupied by the local variables of a subroutine, and so on. Errors occur when the instruction operands are incorrectly considered, such as setting the number of local variables of a subroutine in place of the number of memory words occupied by these variables.

- The generation of code for manipulating value and reference parameters conveys many error possibilities. A few points to be regarded are:

  [1] Procedures `read` and `write` with real formal parameters by value and reference. Also the cases `read(x[1])` (reading a array element) where `x` is a formal parameter by value/reference must be considered.

  [2] In the subroutine call
  `P(x)`
  there are four cases to be considered from the combination of the following options:

  – `x` is a parameter by value/reference.

  – The formal parameter of `P` is by value/reference. This test must also be applied if `x` is a array or an element of a array (e.g. `P(v[1])`).

  [3] All parameter types (value/reference, basic type/array) must be considered on the left side of an attribution: `X:=Y`

- The address of an indexed array (e.g. `a[i,j]`) must be checked for correctness. Such address is calculated in three situations:

  – When an element `a[i,j]` of a array is passed as a parameter by reference.

  – `a[i,j]` is a real parameter by value or is placed in an expression.

  – `a[i,j]` is on the left side of an attribution (`:=`).

  In each of these cases, it must also be checked whether `a[i,j]` is of a basic type (`integer`, `boolean`) or is a array (thus having at least 3 dimensions).

- Prior to invoking a function, enough space for its return value must be allocated on the stack.

- In a typical compiler, the labels are produced in the form of an `L` followed by a number, e.g. `L1`, `L2`, `L3`, ... A compiler routine like `GetLabel` is responsible for providing the next label to be used in the code generation. A possible error is to associate a label declared in the Pascal program as
  `label 2;`
  to the code generation label named `L2`. The correct procedure would be to associate label `2` of the program to a real label generated by `GetLabel`.

3.5 Errors in the Symbol Table

This subsection discusses errors related to the inserting, consulting, or removing information from the data structure used in the compiler Symbol Table (ST).

```
program UpperLower;
var index, NUMBER : integer ;
begin
{ checks if letter cases
  are disregarded }
INDEX := 1 ;
number := 2
end.
```

Figura 4: This program tests whether the compiler considers upper and lower case letter as equivalent

- A reserved word is not considered as such. Conversely, it is also an error to consider non-reserved words like `file`, `real`, `packed`, `string`, `case`, `for`, as reserved words. Such words are not employed in the Pascal dialect used in this article.
- Identifiers lying out of the scope must be eliminated from the ST. Upon terminating the compilation of a subroutine at lexic level N, all subroutines in level N+1, as well as the variables in level N, must be removed from the ST. The subroutines that are visible in level N are the ones already defined either in lower levels or in level N+1.
- The number of variables and subroutines visible at a given time must be limited exclusively by the amount of memory available in the computer.
- The language accepts an unlimited number of array dimensions. A static data structure to store array information would limit that number unecessarily.
- Pascal considers upper and lower case letters in identifiers as identical. A typical compiler would implement this rule as follows:

  [1] An identifier is converted to upper case prior to its insertion into the ST.

  [2] An identifier is converted to upper case before it is searched for on the ST.

  The test shown in Figure 4 checks whether such actions have been performed:

  [1] Variable `index` is declared in lower case and used in upper case to verify action 1. When a variable is declared, it is inserted in the ST. Whenever it appears in the subroutine body, a search is performed upon the ST.

  [2] Variable `NUMBER` is declared in upper case and used in lower case to verify action 2.

- An identifier declared in lexic level N+1 has a higher priority than another one by the same name, declared in a level equal to or lower than N.

```
if character = '{'
then
  repeat
     character = next character;
  until character = '}'
endif
```

Figura 5: Procedure for removing comments

Let us consider that the Symbol Table is implemented as a stack where the identifiers in lexic level N+1 are placed above (the stack grows upwards) the ones in level N. An error occurs if the search for an identifier starts at the bottom and proceeds to the top of the stack. In this case, the identifiers in level N would be considered first.

3.6 Errors in the Lexic Analyser

The lexic analyser is responsible for forming identifiers and numbers to subsequently pass them to the syntax analyser. It also filters out the remarks. The erroneous generation of a token leads to error propagation through the other analysers, thus causing a general compiler error.

- If the recognition of integer numbers is assigned to the syntax analyser, the compiler will possibly accept numbers with spaces between digits. In this case, all of the three attributions below would be considered equivalent and correct:

  ```
  a:= 12 3; a:= 12{A}3; a:= 123;
  ```

- Unclosed remarks should be predicted in order to avoid compiler crashes. Figure 5 shows an incorrect algorithm for recognizing remarks. The correct procedure would be repeat until character=`'{'` or "EndOfFile".
- The compiler must accept identifiers of any size, considering the first 32 characters as significant. Two identifiers will be equal if the first 32 characters are equal. If only the 32nd character is different, then they will be different. Possible errors are:
  - Either the compiler signals an error when it finds a word with over 32 characters, or it places more than 32 characters in a string type variable of ST that has only 32 positions available. This may cause writing on a memory area that was not assigned this task.
  - Only the first 31 characters (instead of 32) of an identifier are considered significant. Accepting more than 32 characters is not considered an error.

```
procedure GetToken()
begin
if character = '{'
then
  skip comment
endif
if character is not valid as identifier,
   number or operand
then Error()
else Gets Token
endif
end
```

Figura 6: Erroneous procedure for identifying comments

- Valid characters in identifiers are letters and numbers, the first character being a letter. The underscore (_) is not permitted.
- A quite common error is not allowing one comment to be followed by another, as in the example below:

  ```
  { comment }   {}
  ```

  An incorrect algorithm to eliminate comments is given in Figure 6.
- The Tab character (`#^I`) is valid and must count as a space. `^Z` is also valid and means "end of file".
- All characters are valid inside a comment, except the closing character, `'}'`.

3.7 Syntax Analiser Errors

This subsection describes examples that test the fidelity of the syntax analiser to the Pascal grammar used. It is illegal:

- A comma not followed by an identifier in the declaration of variables, parameters, or in subroutine calls.

  ```
  var a, b, : integer;
  procedure P( x, y, : integer); ...
  P( a, b, );
  read( a, );
  write( a, );
  ```

  It is important to have `read` or `write` tests, as well as subroutine call tests (`P`). Such pre-defined procedures are in general implemented with their own syntax analiser routines, that is, an error in the `read` analysis does not imply the existence of the same error in `write` or in normal subroutines.
- Absence of identifier/type:

```
procedure ParameterPassage;
begin
{ Have you found it open parenthesis }
while  token  <> close_parenthesis do
  { analyses parameters and generates code }
  if token <> comma then GENERATE_ERROR
  token = GetNextToken
endwhile
{ It should check here whether the number of
  parameters encountered equals the number of
  formal parameters in the subroutine }
end { procedure }
```

Figura 7: Erroneous procedure for parameter passing analysis

  ```
  var  : integer;
  var  i : ;
  ```

- A NULL command (according to the grammar used) is not allowed. The code below is therefore illegal:

  ```
  while true do
    ;  { null command }
  ```

- Operator without an operand:

  ```
  a:= *2;
  ```

3.8 Errors in the Semantic Analyser

This subsection deals with errors in the semantic analyser of a Pascal compiler. Semantic errors are possible even when the syntax analyser is correct. A few critical situations are mentioned below:

- Absence or excess of subroutine parameters. Most likely, the error will occur if the compiler accepts fewer parameters than the subroutine requires. The algorithm in Figure 7 bears such mistake.
- A function `F` (`F:= expression`) can be assigned return values in its body or within its nested subroutines. Apart from these sites, the assignment will be erroneous.
- Whenever a function `F` is assigned an expression (`F:= expression`), the expression must be of the same type as the return value type of `F` given on the header.
- An expression cannot be real parameter for `read` or when the corresponding formal parameter is by reference.
- Arithmetic operators (logical) can only act upon integers (boolean values).[1]

[1] In the Pascal subset used, the bit-to-bit AND or OR on integers does not exist (e.g. 2 and 7).

According to the Pascal syntax diagram, the operators `and` and `*` are in the same priority level, like `or` and `+`. A typical compiler implementation treats equal priority operators in a while loop:

```
t = Term()  { comment }
while token = Plus or token = OR do
  { analyses operand }
     ...
endwhile
```

A possible error would be checking type matching of the first and last parameters only, thus admitting expressions like

```
1 + 3 + true + 4 + 5
```

- The upper and lower limits of an array are subject to the restriction "upper >= lower". Two errors may happen:

  [1] rejecting upper = lower

  [2] accepting upper < lower

- The arguments of `read` and `write` must be integers. Arrays are not permitted.

- Labels defined twice:

```
      99: write(1);
      ...
      99: write(2);
      ...
```

- `Goto` command to a "label"which has been declared but is not defined in the subroutine body where it was declared. Because this error is only flagged at the end of a subroutine compilation, most student-made compilers will not raise it.

- Conversely, a label error will also occur if the label is defined but has not been declared.

- To use procedures on the left side of an attribution (in its body) or in an expression.

- A function call is an expression.

- The words `read` and `write` are not considered reserved words in Pascal, but are treated as special procedures since the number of parameters is not fixed. `read` and `write` are inserted (in a typical implementation) in the Symbol Table and receive special treatment by the compiler routine that analyses procedure calls. An example of such routines is given in Figure 8. Upon analysing the command "`write(1,2,3)`"of Figure 9 , the compiler classifies the word "`write`"as a procedure,[2] and the algorithm of Figure 8 is executed. It tests whether the word string (the procedure name) is equal to the string `"WRITE"`. Since that is true, the procedure found is regarded as the Pascal pseudo-procedure `write` instead of a user-defined procedure. Hence, procedure `write` declared in the program is always masked by Pascal's `write` and can never be used.

```
if  simbolo = "WRITE"
then
  { Analyses Pascal's write }
else
  if simbolo = "READ"
  then
    { Analyses Pascal's read }
  else
    { Analyses user defined procedure }
  endif
endif
```

Figura 8: Erroneous algorithm for procedure analysis

```
program Wr;
  procedure write( n : integer);
  begin
  n:= 1
  end;
begin
write(1,2,3)
end.
```

Figura 9: User-declared procedure `write`

- Circular type definitions. For instance:

```
type  T = T;
```

  The first `T`, right after type, must only be inserted in the Symbol Table after the analysis of the type following "`=`". The definition "`type integer = integer`"is considered legal.

- A variable `read` has been declared, but the program still accepts the Pascal procedure `read`, which is incorrect. The distinction between the procedure and the variable can be made from the token that follows the identifier ("("or "`:=`"), when it begins a command.

```
read (a) ;
read := 12 ;
```

[2]Other classes are : variable, parameter, type, label,...

4 Conclusion

This article has presented guidelines for designing tests on simplified Pascal compilers like the ones built in introductory compilation courses. The guidelines comprehend several topics, such as code generation, lexic analysis, syntax, and semantics. Within each topic, a considerable number of possible compiler errors have been explained.

Acknowledgments. We are thankful to José Carlos Maldonado for the revision of this article and for the suggestions presented.

Referências

[1] Myers, Glenford J. *The Art of Software Testing.* John Wiley & Sons, 1979.

[2] Wirth, Niklaus and Jensen, Kathleen. *Pascal User Manual and Report*, Springer-Verlag, 1985, Third Edition.