Lexical Analyzer

Chris Johnson

October 24, 2009

1 Introduction

Lexical analysis is the first stage of compilation. In it, a program's source is parsed into tokens, each of which is tagged with metadata, indicating how we interpret the token, and checked for satisfaction of constraints.

2 Methodology

In this implementation of a lexical analyzer, we use Haskell, a purely functional language. This presents some challenges which would not be present in procedural languages – for instance, the lack of iteration requires that state be passed through functions[1]. On the other hand, it enforces a logical separation of tasks which, I believe, renders the program more readily adaptable to other languages than a more "mixed" approach would.

The methodology can be described in three stages: tokenization, error-checking, and interpretation. This ordering is conceptual only; in point of fact, the interpretation code is entered first, and calls the tokenizer through the error-checker. However, the tokenizer produces the pieces with which the other stages must work.

2.1 Tokenization

One advantage of the functional paradigm is that NFAs are readily implementable as a series of functions. (By allowing our functions to examine more than one character at a time in choosing their responses, we simulate the backtracking which distinguishes NFAs from DFAs.)

Essentially, then, we begin with a string of characters, and check for five classes of object: whitespace, words (i.e. identifiers, reserved words, and operator words), numbers, operators, and delimeters.

Whitespace, words, and numbers are read greedily, since this is how a human would group them. As a result, we postpone length checks on the words, wishing here to focus purely on parsing tokens in the most sensible way.

[1] It is possible to track state via a state monad. At present, this solution does not seem to confer any particular advantages that function arguments do not.

Addition and multiplication operators, as well as such delimeters as parentheses, square braces and semicolons, being single characters, are immediately tokenized.

The minus ('-'), colon (':'), and period ('.') characters are exceptions to the above rule, as they are ambiguous. A minus or a period preceding a digit lead to the numeric parser (to the whole and fractional parsers, respectively). A colon preceding an equals ('=') becomes part of an assignment operator, rather than a delimeter.

Relational operators are grouped in pairs of two where possible, though, of course, a single relational operator is recognized as such. However, this does present the possibility of invalid sequences like "=>" or "><", so unrecognized pairs cause a lexical error.

Only the most basic error-checking is implemented here. For example, characters that do not appear in the language will not be matched by our NFA, so it is only appropriate to throw up a lexical error. Token length is not considered. The only other lexical error to appear is an unknown relational operator.

Lexical errors are returned as tokens.

2.2 Pre-Processing

The next stage is to preprocess the tokens, in preparation for generating the symbol table, and listing and token files.

Here we simply filter out whitespace, distinguish identifiers from reserved words, and enforce constraints on numbers and identifiers.

All remaining lexical errors are caught in this phase, and returned as tokens for the interpretation stage to inform the programmer.

2.3 Interpretation

We here use interpretation in the usual sense of drawing information out of data, rather than that of executing code without a separate compilation process.

The reserved words file is read in, followed by the source file. Each line of the source file is tokenized and pre-processed. The resulting list of tokens is then further processed to build a symbol table and replace identifiers with symbol table references. (The symbol table is implemented as a list of identifiers, and the references are simply indices.)

Each line is then written to the listing file, along with any lexical errors caught in that line. All tokens (including lexical errors) are written to the token file.

As this is also the entry point of IO, extra-long lines are also caught and reported in this stage.

2.4 Deviations

There are a few minor ways in which this implementation deviates from the project specification, and I feel they are worth noting.

First, of course, I used Haskell instead of Java. I feel this leads to a conceptual coherence that can easily be lost in Java. By eliminating the *possibility* of side effects, each stage either does one job or becomes hopelessly unwieldy. This improves the adaptability of the compiler: once the desired change is identified, only specific pieces need be modified.

The 72 character line limit is not enforced, but rather gives a warning in the listing file. It seemed somewhat absurd to halt compilation for a non-issue.

The reserved word file merely lists the reserved words; I believe nothing particularly useful is achieved by adding on numbers for token type and attribute, as any speed they may confer to the compiler is at the cost of clarity.

Finally, the formatting of the token and listing files deviates heavily from the examples given. The reasoning is simple: Haskell's data type system allows all of the necessary information to be contained directly in an object, and can readily be parsed by another Haskell program with the same types.

3 Conclusions

In this project, I implemented a fairly straightforward lexical analyzer, utilizing multiple conceptual stages of analysis. Using Haskell's pattern-matching, I was able to generate clear code that stays faithful to the conceptual framework of NDFAs under which we were working.

A Error-Free Test Program

This program should test every production with no errors.

A.1 Source

```
program example(input, output);
var x: integer;
var y: integer;
var z: real;
var a: array[0 .. 1] of integer;

function gcd(a: integer; b: integer): integer;
        begin
                if b = 0
                then
                        gcd := a
                else
                        gcd := gcd(b, a mod b)
        end;

function lcm(a: integer; b: integer): integer;
        var c: integer;
        function useless: integer;
                begin
                end;
        begin
                c := gcd(a, b);
```

```
                lcm := a * b / c
        end;

begin
        x := gcd(123,456);
        y := lcm(12,34);
        z := 3 * -1.123e45;

        i := 10;
        while i > 0 do
                begin
                        i := i - 1
                end;

        if not (i = 0) then
                i := 1
        else
                i := 0;

        a[0] := 1;
        a[1] := a[0]
end.
```

A.2 Listing File

```
1:      program example(input, output);
2:      var x: integer;
3:      var y: integer;
4:      var z: real;
5:      var a: array[0 .. 1] of integer;
7:      function gcd(a: integer; b: integer): integer;
8:              begin
9:                      if b = 0
10:                      then
11:                             gcd := a
12:                      else
13:                             gcd := gcd(b, a mod b)
14:             end;
16:     function lcm(a: integer; b: integer): integer;
17:             var c: integer;
18:             function useless: integer;
19:                     begin
20:                     end;
21:             begin
22:                     c := gcd(a, b);
```

```
23:                     lcm := a * b / c
24:             end;
26:     begin
27:             x := gcd(123,456);
28:             y := lcm(12,34);
29:             z := 3 * -1.123e45;
31:             i := 10;
32:             while i > 0 do
33:                     begin
34:                             i := i - 1
35:                     end;
37:             if not (i = 0) then
38:                     i := 1
39:             else
40:                     i := 0;
42:             a[0] := 1;
43:             a[1] := a[0]
44:     end.
```

A.3 Token File

```
Line 1: 'program'
Line 1: symbol table entry 1
Line 1: '('
Line 1: symbol table entry 2
Line 1: ','
Line 1: symbol table entry 3
Line 1: ')'
Line 1: ';'
Line 2: 'var'
Line 2: symbol table entry 4
Line 2: ':'
Line 2: 'integer'
Line 2: ';'
Line 3: 'var'
Line 3: symbol table entry 5
Line 3: ':'
Line 3: 'integer'
Line 3: ';'
Line 4: 'var'
Line 4: symbol table entry 6
Line 4: ':'
Line 4: 'real'
Line 4: ';'
Line 5: 'var'
```

```
Line 5: symbol table entry 7
Line 5: ':'
Line 5: 'array'
Line 5: '['
Line 5: '0'
Line 5: '..'
Line 5: '1'
Line 5: ']'
Line 5: 'of'
Line 5: 'integer'
Line 5: ';'
Line 7: 'function'
Line 7: symbol table entry 8
Line 7: '('
Line 7: symbol table entry 7
Line 7: ':'
Line 7: 'integer'
Line 7: ';'
Line 7: symbol table entry 9
Line 7: ':'
Line 7: 'integer'
Line 7: ')'
Line 7: ':'
Line 7: 'integer'
Line 7: ';'
Line 8: 'begin'
Line 9: 'if'
Line 9: symbol table entry 9
Line 9: '='
Line 9: '0'
Line 10: 'then'
Line 11: symbol table entry 8
Line 11: ':='
Line 11: symbol table entry 7
Line 12: 'else'
Line 13: symbol table entry 8
Line 13: ':='
Line 13: symbol table entry 8
Line 13: '('
Line 13: symbol table entry 9
Line 13: ','
Line 13: symbol table entry 7
Line 13: 'mod'
Line 13: symbol table entry 9
Line 13: ')'
```

```
Line 14: 'end'
Line 14: ';'
Line 16: 'function'
Line 16: symbol table entry 10
Line 16: '('
Line 16: symbol table entry 7
Line 16: ':'
Line 16: 'integer'
Line 16: ';'
Line 16: symbol table entry 9
Line 16: ':'
Line 16: 'integer'
Line 16: ')'
Line 16: ':'
Line 16: 'integer'
Line 16: ';'
Line 17: 'var'
Line 17: symbol table entry 11
Line 17: ':'
Line 17: 'integer'
Line 17: ';'
Line 18: 'function'
Line 18: symbol table entry 12
Line 18: ':'
Line 18: 'integer'
Line 18: ';'
Line 19: 'begin'
Line 20: 'end'
Line 20: ';'
Line 21: 'begin'
Line 22: symbol table entry 11
Line 22: ':='
Line 22: symbol table entry 8
Line 22: '('
Line 22: symbol table entry 7
Line 22: ','
Line 22: symbol table entry 9
Line 22: ')'
Line 22: ';'
Line 23: symbol table entry 10
Line 23: ':='
Line 23: symbol table entry 7
Line 23: '*'
Line 23: symbol table entry 9
Line 23: '/'
```

```
Line 23: symbol table entry 11
Line 24: 'end'
Line 24: ';'
Line 26: 'begin'
Line 27: symbol table entry 4
Line 27: ':='
Line 27: symbol table entry 8
Line 27: '('
Line 27: '123'
Line 27: ','
Line 27: '456'
Line 27: ')'
Line 27: ';'
Line 28: symbol table entry 5
Line 28: ':='
Line 28: symbol table entry 10
Line 28: '('
Line 28: '12'
Line 28: ','
Line 28: '34'
Line 28: ')'
Line 28: ';'
Line 29: symbol table entry 6
Line 29: ':='
Line 29: '3'
Line 29: '*'
Line 29: '-1.123e45'
Line 29: ';'
Line 31: symbol table entry 13
Line 31: ':='
Line 31: '10'
Line 31: ';'
Line 32: 'while'
Line 32: symbol table entry 13
Line 32: '>'
Line 32: '0'
Line 32: 'do'
Line 33: 'begin'
Line 34: symbol table entry 13
Line 34: ':='
Line 34: symbol table entry 13
Line 34: '-'
Line 34: '1'
Line 35: 'end'
Line 35: ';'
```

```
Line 37: 'if'
Line 37: 'not'
Line 37: '('
Line 37: symbol table entry 13
Line 37: '='
Line 37: '0'
Line 37: ')'
Line 37: 'then'
Line 38: symbol table entry 13
Line 38: ':='
Line 38: '1'
Line 39: 'else'
Line 40: symbol table entry 13
Line 40: ':='
Line 40: '0'
Line 40: ';'
Line 42: symbol table entry 7
Line 42: '['
Line 42: '0'
Line 42: ']'
Line 42: ':='
Line 42: '1'
Line 42: ';'
Line 43: symbol table entry 7
Line 43: '['
Line 43: '1'
Line 43: ']'
Line 43: ':='
Line 43: symbol table entry 7
Line 43: '['
Line 43: '0'
Line 43: ']'
Line 44: 'end'
Line 44: '.'
```

B Error-imbued Test Program

This test program is nearly identical to the previous one, except designed to cause every lexical error.

B.1 Source

```
program example(input, output);
var x: integer;
```

```
var y: integer;
var z: real;
var a: array[0 .. 3] of integer;
var thisidisreallytoolong: integer;

><

function gcd(a: integer; b: integer): integer;
begin
if b = 0
then
gcd := a
else
gcd := gcd(b, a mod b)
end;

function lcm(a: integer; b: integer): integer;
var c: integer;
function useless: integer;
begin
end;
begin
c := gcd(a, b);
lcm := a * b / c
end;

begin
x := gcd(123,456);
y := lcm(12,34);
z := 3 * -1.123e45;

i := 10;
while i > 0 do
begin
i := i - 1
end;

if not (i = 0) then
i := 1
else
i := 0;

a[0] := 112358132134;
a[1] := 2718281828459e-12;
a[2] := .314159265358979e1;
```

```
a[3] := 1.23e456
end.
```

B.2 Listing File

```
1: program example(input, output);
2: var x: integer;
3: var y: integer;
4: var z: real;
5: var a: array[0 .. 3] of integer;
6: var thisidisreallytoolong: integer;
Lexical Error (Extra Long Identifier): thisidisreallytoolong
8: ><
Lexical Error (Unrecognized Symbol): ><
10: function gcd(a: integer; b: integer): integer;
11: begin
12: if b = 0
13: then
14: gcd := a
15: else
16: gcd := gcd(b, a mod b)
17: end;
19: function lcm(a: integer; b: integer): integer;
20: var c: integer;
21: function useless: integer;
22: begin
23: end;
24: begin
25: c := gcd(a, b);
26: lcm := a * b / c
27: end;
29: begin
30: x := gcd(123,456);
31: y := lcm(12,34);
32: z := 3 * -1.123e45;
34: i := 10;
35: while i > 0 do
36: begin
37: i := i - 1
38: end;
40: if not (i = 0) then
41: i := 1
42: else
43: i := 0;
45: a[0] := 112358132134;
```

```
Lexical Error (Extra Long Integer): 112358132134
46: a[1] := 2718281828459e-12;
Lexical Error (Extra Long Whole Part): 2718281828459e-12
47: a[2] := .314159265358979e1;
Lexical Error (Extra Long Fractional Part): .314159265358979e1
48: a[3] := 1.23e456
Lexical Error (Extra Long Exponent): 1.23e456
49: end.
```

B.3 Token File

```
Line 1: 'program'
Line 1: symbol table entry 1
Line 1: '('
Line 1: symbol table entry 2
Line 1: ','
Line 1: symbol table entry 3
Line 1: ')'
Line 1: ';'
Line 2: 'var'
Line 2: symbol table entry 4
Line 2: ':'
Line 2: 'integer'
Line 2: ';'
Line 3: 'var'
Line 3: symbol table entry 5
Line 3: ':'
Line 3: 'integer'
Line 3: ';'
Line 4: 'var'
Line 4: symbol table entry 6
Line 4: ':'
Line 4: 'real'
Line 4: ';'
Line 5: 'var'
Line 5: symbol table entry 7
Line 5: ':'
Line 5: 'array'
Line 5: '['
Line 5: '0'
Line 5: '..'
Line 5: '3'
Line 5: ']'
Line 5: 'of'
Line 5: 'integer'
```

```
Line 5: ';'
Line 6: 'var'
Line 6: Lexical Error (Extra Long Identifier): thisidisreallytoolong
Line 6: ':'
Line 6: 'integer'
Line 6: ';'
Line 8: Lexical Error (Unrecognized Symbol): ><
Line 10: 'function'
Line 10: symbol table entry 8
Line 10: '('
Line 10: symbol table entry 7
Line 10: ':'
Line 10: 'integer'
Line 10: ';'
Line 10: symbol table entry 9
Line 10: ':'
Line 10: 'integer'
Line 10: ')'
Line 10: ':'
Line 10: 'integer'
Line 10: ';'
Line 11: 'begin'
Line 12: 'if'
Line 12: symbol table entry 9
Line 12: '='
Line 12: '0'
Line 13: 'then'
Line 14: symbol table entry 8
Line 14: ':='
Line 14: symbol table entry 7
Line 15: 'else'
Line 16: symbol table entry 8
Line 16: ':='
Line 16: symbol table entry 8
Line 16: '('
Line 16: symbol table entry 9
Line 16: ','
Line 16: symbol table entry 7
Line 16: 'mod'
Line 16: symbol table entry 9
Line 16: ')'
Line 17: 'end'
Line 17: ';'
Line 19: 'function'
Line 19: symbol table entry 10
```

```
Line 19: '('
Line 19: symbol table entry 7
Line 19: ':'
Line 19: 'integer'
Line 19: ';'
Line 19: symbol table entry 9
Line 19: ':'
Line 19: 'integer'
Line 19: ')'
Line 19: ':'
Line 19: 'integer'
Line 19: ';'
Line 20: 'var'
Line 20: symbol table entry 11
Line 20: ':'
Line 20: 'integer'
Line 20: ';'
Line 21: 'function'
Line 21: symbol table entry 12
Line 21: ':'
Line 21: 'integer'
Line 21: ';'
Line 22: 'begin'
Line 23: 'end'
Line 23: ';'
Line 24: 'begin'
Line 25: symbol table entry 11
Line 25: ':='
Line 25: symbol table entry 8
Line 25: '('
Line 25: symbol table entry 7
Line 25: ','
Line 25: symbol table entry 9
Line 25: ')'
Line 25: ';'
Line 26: symbol table entry 10
Line 26: ':='
Line 26: symbol table entry 7
Line 26: '*'
Line 26: symbol table entry 9
Line 26: '/'
Line 26: symbol table entry 11
Line 27: 'end'
Line 27: ';'
Line 29: 'begin'
```

```
Line 30: symbol table entry 4
Line 30: ':='
Line 30: symbol table entry 8
Line 30: '('
Line 30: '123'
Line 30: ','
Line 30: '456'
Line 30: ')'
Line 30: ';'
Line 31: symbol table entry 5
Line 31: ':='
Line 31: symbol table entry 10
Line 31: '('
Line 31: '12'
Line 31: ','
Line 31: '34'
Line 31: ')'
Line 31: ';'
Line 32: symbol table entry 6
Line 32: ':='
Line 32: '3'
Line 32: '*'
Line 32: '-1.123e45'
Line 32: ';'
Line 34: symbol table entry 13
Line 34: ':='
Line 34: '10'
Line 34: ';'
Line 35: 'while'
Line 35: symbol table entry 13
Line 35: '>'
Line 35: '0'
Line 35: 'do'
Line 36: 'begin'
Line 37: symbol table entry 13
Line 37: ':='
Line 37: symbol table entry 13
Line 37: '-'
Line 37: '1'
Line 38: 'end'
Line 38: ';'
Line 40: 'if'
Line 40: 'not'
Line 40: '('
Line 40: symbol table entry 13
```

```
Line 40: '='
Line 40: '0'
Line 40: ')'
Line 40: 'then'
Line 41: symbol table entry 13
Line 41: ':='
Line 41: '1'
Line 42: 'else'
Line 43: symbol table entry 13
Line 43: ':='
Line 43: '0'
Line 43: ';'
Line 45: symbol table entry 7
Line 45: '['
Line 45: '0'
Line 45: ']'
Line 45: ':='
Line 45: Lexical Error (Extra Long Integer): 112358132134
Line 45: ';'
Line 46: symbol table entry 7
Line 46: '['
Line 46: '1'
Line 46: ']'
Line 46: ':='
Line 46: Lexical Error (Extra Long Whole Part): 2718281828459e-12
Line 46: ';'
Line 47: symbol table entry 7
Line 47: '['
Line 47: '2'
Line 47: ']'
Line 47: ':='
Line 47: Lexical Error (Extra Long Fractional Part): .314159265358979e1
Line 47: ';'
Line 48: symbol table entry 7
Line 48: '['
Line 48: '3'
Line 48: ']'
Line 48: ':='
Line 48: Lexical Error (Extra Long Exponent): 1.23e456
Line 49: 'end'
Line 49: '.'
```