Simple language

— The only data type is integer.
— All identifiers are implicitly declared and are no longer than 32 characters. Identifiers must begin with a letter and are composed of letters, digits, and underscores.
— Literals are strings of digits.
— Comments begin with — — and end at the end of the current line.
— Statement types are as follows:
Assignment:
Id := Expression;
Expression is an infix expression constructed from identifiers, literals, and the operators + and — ; parentheses are also allowed.
Input / Output:
read (List of Id's);
write (List of Expressions);
— begin, end, read, and write are reserved words.
— Each statement is terminated by a semicolon (;). The body of a program is delimited by begin and end.
— A blank is appended to the right end of each source line; thus tokens may not extend across line boundaries.

Example

```
BEGIN — —SOMETHING UNUSUAL
READ(A1, New_A, D, B);
C:= A1 + (New_A — D) — 75;
New_C:=((B — (7)+(C+D))) — (3 — A1); — — STUPID FORMULA
WRITE (C, A1+New_C);
— — WHAT ABOUT := B+D;
END
```

The Structure of Simple Compiler

— The scanner reads a source program from a text file and produces a stream of token representations. So that no actual stream need not exist at any time, the scanner is actually a function that produces token representations one at a time when called by the parser.
— The parser processes tokens until it recognizes a syntactic structure that requires semantic processing. It then takes a direct call to a semantic routine. Some of these semantic routines use token representation information in their processing.
— The semantic routines produce output in assembly language for a simple virtual machine. Thus the compiler structure includes no optimizer, and code generation is done by direct calls to appropriate routines from the set of semantic routines.
— The symbol table is used only by the semantic routines.

Some Pseudocode Constructs

```
while <expr>
loop
        <statement>
        <statement>
        . . .
        <statement>
end loop;

case <expr> is
        when <expr> | <expr> | . . . =>
                <list of statements>;
                . . .
        when <expr> | <expr> | . . . =>
                <list of statements>;
        when others =>
                <list of statements>;
end case;

if <expr> then
        statement>;
else
        <statement>;
end if;

function <name> (<formal param list>)
        return <type name> is
        Type, variable, constant, and subprogram declarations
begin
        Statement list
end;
```

A Simple Scanner

The scanner will be a function that takes no arguments and returns Token values.

```
type Token is
(BeginSym, EndSym, ReadSym, WriteSym, Id, IntLiteral, LParen,
RParen, SemiColon, Comma, AssignOp, PlusOp, MinusOp, EofSym);
```

TokenBuffer: String

function Scanner return Token;
The following routines will be used:

Read(C) — The next input character is read; error if Eof is true.

func. Inspect — The next input character is returned, but input is not advanced; error if Eof is true.

Advance — The next input character is removed, but not returned; no effect at end of file.

func. Eof — True at the end of file.

BufferChar — Adds its argument to a character buffer called TokenBuffer. This buffer is visible to any part of the compiler, and always contains the text of the most recently scanned token. The content of TokenBuffer will be used particularly by semantic routines. Of course, the characters of this buffer also are used by CheckReserved to determine whether a token that looks like an identifier is actually a reserved word.

ClearBuffer — Reset the buffer TokenBuffer to the empty string

func. CheckReserved—Takes the identifiers as they are recognized and returns the proper token class (either Id or some reserved word).

```
function Scanner return Token is
    begin
        ClearBuffer;
        if Eof then
            return EofSym
        else
            while not Eof
            loop
                Read(CurrentChar)
                case CurrentChar is
                    when . . . =>. . . .
                    . . . . . .
                end case;
            end loop;
            return EofSym
        end if
    end Scanner
```

```
function Scanner return Token is
begin
        ClearBuffer;
        if Eof then
                return EofSym
        else
                while not Eof
                loop
                        Read(CurrentChar)
                        case CurrentChar is
                                . . . —— Code to recognize ‘ ‘, Tab, Eol
                                . . . —— Code to recognize identifiers and reserved words
                                . . . —— Code to recognize integer literals
                                . . .—— Code to recognize delimiters
                                . . .—— Code to recognize operators
                                . . .—— Code to recognize comments
                                when others => LexicalError(CurrentChar);
                        end case;
                end loop;
                return EofSym
        end if
end Scanner
```

Scanner Loop
to Recognize Identifiers, Reserved Words, and Literals

```
while not Eof
loop
        Read(CurrentChar)
        case CurrentChar is
                when ‘ ‘| Tab| Eol =>
                        null
                when ‘A’..’Z’| ‘a’.. ‘z’=>
                loop
                        case Inspect is
                                when ‘A’..’Z’| ‘a’.. ‘z’| ‘0’..’9’| ‘_’ => Advance;
                                when others => return CheckReserved
                        end case;
                end loop;
                when ‘0’..’9’ =>
                loop
                        case Inspect is
                                when ‘0’..’9’ => Advance;
                                when others => return IntLiteral;
                        end case
                end loop
                when others => LexicalError(CurrentChar);
        end case;
end loop;
```

Scanner Loop
to Recognize Operators, Comments and Delimiters

```
type Token is (BeginSym, EndSym, ReadSym, WriteSym, Id, IntLiteral, LParen,
RParen, SemiColon, Comma, AssignOp, PlusOp, MinusOp, EofSym);

function Scanner return Token;

while not Eof
loop
        Read(CurrentChar)
        case CurrentChar is
                . . . —— Code to recognize ‘ ‘, Tab, Eol
                . . . —— Code to recognize identifiers
                . . . —— Code to recognize integer literals
                when ‘(‘ => return LParen;
                when ‘)‘ => return RParen;
                when ‘;‘ => return SemiColon;
```

```
            when ‘,‘ => return Comma;
            when ‘+‘ => return PlusOp;
            when ‘:’ =>

            if Inspect = ‘=‘ then
                  Advance;
                  return AssignOp;
            else
                  LexicalError(Inspect);
            end if;
            when ‘—’ =>
                  if Inspect = ‘—’ then
                        Read(CurrentChar);
                        while CurrentChar ≠ Eol
                        loop
                              Read(CurrentChar)
                        end loop;
                  else
                        return MinusOp;
                  end if
            when others => LexicalError(CurrentChar);
      end case;
end loop
```

Complete Scanner Function for Simple scanner
with recognition of reserved words and Eof

functions

CheckReserved — Takes the identifiers as they are recognized and returns the proper token class (either Id or some reserved word).

BufferChar(CurrentChar) — Adds its argument to a character buffer called TokenBuffer

ClearBuffer — Resets the buffer to the empty string

type Token is (BeginSym, EndSym, ReadSym, WriteSym, Id, IntLiteral, LParen, RParen, SemiColon, Comma, AssignOp, PlusOp, MinusOp, EofSym);

```
function Scanner return Token is
begin
        ClearBuffer;
        if Eof then
                return EofSym
        else
        while not Eof
        loop
                Read(CurrentChar)
                case CurrentChar is
                        when ‘ ’| Tab| Eol => null
                        when ‘A’..‘Z’| ‘a’.. ‘z’=> BufferChar(CurrentChar);
                        loop
                        case Inspect is when ‘A’..’Z’| ‘a’.. ‘z’| ‘0’..’9’| ‘_’ =>
                                                                BufferChar(CurrentChar);
                                                                Advance;
                        when others => return CheckReserved;
                end case
        end loop
        when ‘0’..’9’ => BufferChar(CurrentChar);
        loop
                case Inspect is
                        when ‘0’..’9’ => BufferChar(CurrentChar);
                                        Advance;
                        when others =>
                                return IntLiteral;
                end case
        end loop
        when ‘(‘ => return LParen;
        when ‘)‘ => return RParen;
        when ‘;‘ => return SemiColon;
        when ‘,‘ => return Comma;
        when ‘+‘=> return PlusOp;
        when ‘:’ =>
                if Inspect = ‘=‘ then
                        Advance;
                        return AssignOp;
                else
                        LexicalError(Inspect);
                end if;
```

```
        when '—' =>
                if Inspect = '—' then
                        Read(CurrentChar);
                        while CurrentChar =/ Eol
                        loop
                                Read(CurrentChar)
                        end loop;
                else
                        return MinusOp;
                end if
        when others => LexicalError(CurrentChar);
end case;
end loop;
return EofSym
end if
end Scanner
```