Additional notes on Language Processors

Jose M Garrido

Summary of Lexical Analysis

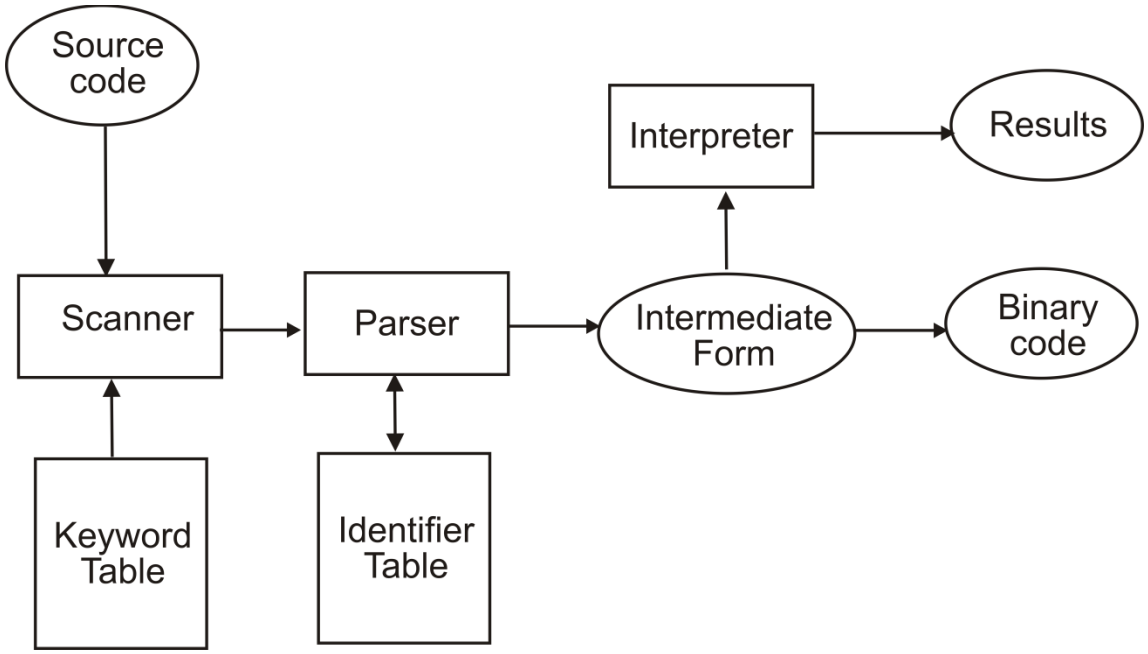
The scanner has the following goals:

- Recognize the next token in the source line
- Recognize if the token is a keyword in the language by a lookup in the keyword table, return the number code for the keyword
- If not a keyword, return the numeric code for the token: integer identifier, float identifier, string identifier, boolean identifier, keyword (one number code for each keyword), a constant (one number code for each type), string literal.
- Store the name of the identifier in a global variable
- Store the value of each constant or literal in a global variable. I recommend a global variable for every type of constant or literal.
- Store the number of the source line scanned in a variable.

A compiler and an interpreter have the same front modules: a scanner and a parser. The interpreter will execute a statement after the parser has recognized it. A compiler will carry out some optimization and generate the binary code of the program.

An internal (or intermediate) form of the parsed program may be used. This has the advantage of promoting more modularity in the overall design and implementation of the language processor. From this intermediate code, the language processor may perform immediate execution of the various operations that have been placed in the intermediate form, or this intermediate form of the program may be stored as a file for later interpretation.

A compiler will take the intermediate form will perform some optimization and generate the binary code of the program.



Symbol Table

I recommend using two separate tables:

- The keyword table, handled by the scanner
- The identifier table, handled by the parser. This table includes several attributes for every identifier, this includes the current value of the identifier. Constants can also be stored in this table.

Two functions are used to manipulate these tables:

- int insert (string name, int token), used with the identifier table (only)
- int lookup (string name), used with both tables

The two functions return the index value of the entry in the table.

Intermediate Forms

Reverse Polish Notation

Polish notation, also known as postfix or suffix notation, is used to represent arithmetic or logical expressions in a manner that specifies simply and exactly the order in which operators are to be evaluated. No parentheses are needed. Operators appear immediately after the operands.

The parser and with semantic functions can convert arithmetic expressions in infix notation to postfix notation.

In reverse Polish notation the operators follow their operands; for instance, to add 3 and 4, one would write “3 4 +” rather than “3 + 4”. If there are multiple operations, the operator is given immediately after its second operand; so the expression written “3 – 4 + 5” in conventional notation would be written “3 4 – 5 +” in RPN: 4 is first subtracted from 3, then 5 added to it.

An advantage of RPN is that it obviates the need for parentheses that are required by infix. While “3 – 4 * 5” can also be written “3 – (4 * 5)”, that means something quite different from “(3 – 4) * 5”. In postfix, the former could be written “3 4 5 * -“, which unambiguously means “3 (4 5 *) -” which reduces to “3 20 -“; the latter could be written “3 4 – 5 *” (or 5 3 4 – *, if keeping similar formatting), which unambiguously means “(3 4 -) 5 *”.

Extended RPN

Each operator is represented by an integer code. Each type of operand is also represented this way. An operand might occupy two locations: the first contains an integer different from any integer representing an operator, while the second contains the operand itself. The types of operands are simple names (of variables, procedures, etc), constants. The operand field can also specify indirect addressing. This is useful for subscripted variables.

Operands must be immediately followed by their operators. An assignment statement will be written

<var> <expr> ASSGNOP

After evaluation, both operands must be deleted from the stack, since there is no resulting value. Conditional branches have the form:

<operand_1> <operand_2> BP

Here, the first operand is an arithmetic value, the second is the number or location of a symbol in the polish string. If the first operand is positive, then the next symbol to scan is the one designated by the second operand; otherwise proceed as usual. Other similar operations are branch on minus, on zero, etc.

A conditional statement such as:

IF <expr> THEN <statement_1> ELSE <statement_2>

Would be written as:

<expr> <op_1> BZ <statement_1> <op_2> BR <statement_2>

- <op_1> is a constant and is the number of the symbol or location beginning <statement_2>.
- BZ is an operator with two operands -- <expr> and <op_1> and performs the action: if (and only if) <expr> is zero then change the order in which the symbols are evaluated by beginning at the one numbered <op_1>.
- <op_2> is the number of the symbol following <statement_2>
- BR is an operator with one operand -- <op_2> and it will change the order in which symbols are evaluated by beginning at the one numbered <op_2>.

Interpreters

The interpreter performs two general operations:

- Perform lexical and syntax checking. Possibly convert the source program into an intermediate form of the program.
- Execute the intermediate form program.

An additional table or stack is used to store the values of all variables defined in the source program. Depending on the variable types used this might be organized in a simple manner as separate tables, one for each type.

For the internal form, an extended form of RPN or reverse polish notation (postfix) is often used. This internal form of the program is stored in an integer array, P. When executing the internal form, an index of this array is used as an "instruction counter", and a stack is used by the interpreter.

I have found the following scheme useful for simple interpreters. The array P with the internal form stores the code of an operand or operator. If it is an identifier or constant, the next element stores the current value of the identifier or constant.

During interpretation, an integer variable, px, is initially 1 and represents the index of array P of the symbol currently being processed. In this way, px is a "program counter". A stack (initially empty) is used by the interpreter to execute the program in extended RPN form. Use the following example to guide you for the integer codes for operators and operands:

5001 INT_CONSTANT

5002 R_CONSTANT

5003 ASSIGN_OP

5004 IF_OP

5005 ELSE_SK

5006 PLUS_OP

5007 UNARY_MINUS

5008 MINUS_OP

5009 MULT_OP

5010 DIV_OP

5011 ENDIF_SK

5012 IDENTIFIER

These numeric codes should not be in conflict with other numeric codes in the language processor. The main part of the interpreter is implemented with a case statement. Each case alternative for each of the possible different operators and types of operands in extended RPN. Assume that array P is of size LAST. A partial implementation of a simplified interpreter is shown next in C++ with embedded pseudo-code.

```
while (px < LAST) {
switch(px) {
     case IDENTIFIER:
        px++;
        take P(px);
        push the value of the identifier on the stack S;
        break;
    case INT_CONSTANT:
        px++;
        take P(px);
        push the value of the constant on the stack S;
        break;
   case ASSIGN_OP:
       pop the last two items from the stack S;
       store the value of the second item popped into the identifier value (first item popped)
      break;
   …..   other case alternatives
  }  // end case
  px++;   // advance to next symbol in P
}  // end while
```

Abstract Stack Machine

An internal form of a program is the code for an abstract stack machine. It has its own instruction set, address space, and all arithmetic operations are performed on values on a stack.

The abstract machine code for an an arithmetic expression simulates the evaluation of a postfix representation for that expression using a stack.

Example of instructions:

Push value

Push value of ident

Push address of ident

Add op (pop the two values on top of stack, add them and push results on the stack)

Assignment op (pop value on top, pop address on top, store the value on the address)

For the infix expression: x = 97 * y, the conversion is:

Push add x

Push 97

Push y

+

Assign op (=)