NOTES ON PROJECT

CS4308 Concepts of Programming Languages

Notes 2

Dr. José M. Garrido
Department of Computer Science

Summer 2017

College of Computing and Software Engineering
Kennesaw State University

1   General Structure of a Language Processor

A language processor is a software implementation of a language. The two general forms of a language processor are:

- A compiler, which will take the entire source program, recognize the statements, check for errors, carry out some optimization and generate the binary code of the program.

- An interpreter, which will recognize each statement, check for errors and execute the statement.

A compiler and an interpreter have the same front modules: a scanner and a parser.

For an interpreter, 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. Figure 1 illustrates the general structure of an interpreter.

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

2   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.
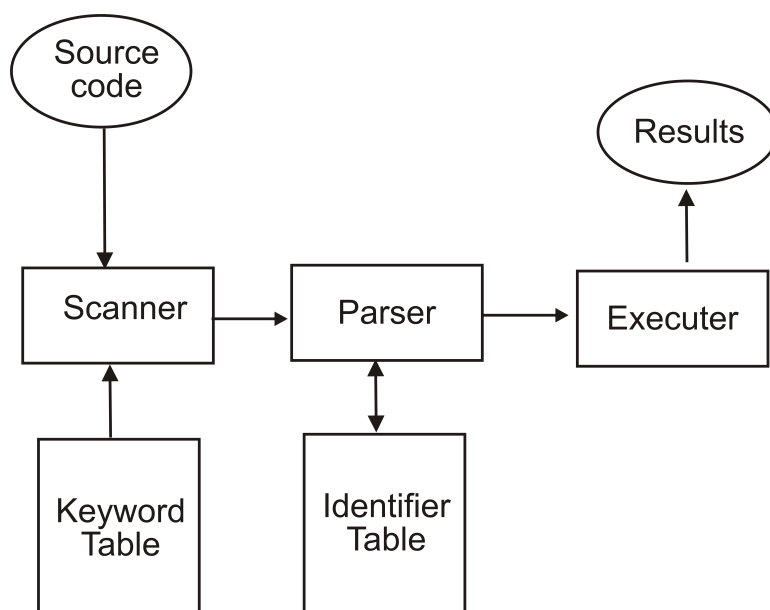
Figure 1: General Structure of an Interpreter

- Store the number of the source line scanned in a variable.

2.1 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:

1. `int insert (string name, int token)`, used with the identifier table (only)

2. `int lookup (string name)`, used with both tables

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

© 2017 J. M. Garrido

3 The Parser

The parser recognizes complete statements using the lexemes from the scanner. Therefore, The parser calls the scanner every time it seems another lexeme.

The scanner returns a numeric code for the type of lexeme (keyword, identifier, operator, etc). The actual value is returned via a global variable. For example the scanner recognized an identifier, so it returns the numeric code for identifiers, and the actual identifier is stored in a string global variable. Similar for a numeric literal, and a string. The parser must display the complete statement that it recognized.

Use the following example to guide you for the integer codes for operators and operands:

```
5001 INT_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
```

4 Intermediate Forms (Optional)

4.1 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 "34+" rather than "+4". If there are multiple operations, the operator is given immediately after its second operand; so the expression written "34 + 5" in conventional notation would be written "345+" 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.

4.2   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

$$\langle var \rangle \langle expr \rangle ASSGNOP$$

After evaluation, both operands must be deleted from the stack, since there is no resulting value.

4.3   Interpreters

The interpreter performs two general operations:

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

2. 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.

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 executer module 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 popp
        break;
    ..    other case alternatives
   } // end case
   px++;  // advance to next symbol in P
} // end while
```

4.4   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 (=)
```