**module** INTERPRETER

**exports**

**procedure** JAZ (INPUT\_FILE: **in file of** char; OUTPUT\_FILE: **out file of** char);

JAZ is called to interpret the program stored in INPUT\_FILE and output the result to OUTPUT\_FILE.

**implementation**

ANALYZER handles syntactic and lexical analysis and produces a SYMBOL\_TABLE. CODE\_PARSER tokenizes the code. MAIN acts as job coordinator.

**is composed of** ANALYZER, SYMBOL\_TABLE, CODE\_PARSER, MAIN

**end** INTERPRETER

**module** MAIN

**uses** ANALYZER, CODE\_PARSER, STACK\_MANIPULATION, CONTROL\_FLOW, ARITHMETIC\_OPERATORS, RELATIONAL\_OPERATORS, LOGICAL\_OPERATORS, OUTPUT

**exports**

**procedure** JAZ (INPUT\_FILE: **in file of** char; OUTPUT\_FILE: **out file of** char);

JAZ is called to interpret the program stored in INPUT\_FILE and output the result to OUTPUT\_FILE.

**implementation**

MAIN will first call CODE\_PARSER, then ANALYZER, then it will execute the code.

**end** MAIN

**module** CODE\_PARSER

**exports**

**type** CODE: array (1...1000) (1,2) **of** string;

**procedure** PARSE (INPUT\_FILE: **in file of** char);

PARSE takes INPUT\_FILE and generates CODE which contains the code all nicely parsed into an array. The SYMBOL\_TABLE will be properly filled out at the conclusion.

**implementation**

The code from the input file will be read line-by-line. It will parse each line, which will be a string, on the first space in the line. In terms of a 2D array, the first line, first token will be stored at array[0][0] and the first line, second token will be stored at array[0][1], so on and so forth until the entire input file is parsed.

**end** CODE\_PARSER

**module** ANALYZER

**uses** SYMBOL\_TABLE

**exports**

**procedure** PUT (ID: **in** IDENTIFIER; DESCR: **in** DESCRIPTOR);

PUT adds a variable to the SYMBOL\_TABLE.

**procedure** GET (ID: **in** IDENTIFIER; LEVEL: **in** INTEGER; DESCR: **out** DESCRIPTOR);

GET retrieves the value stored in a variable.

**procedure** INIT (MAX\_DEPTH: **in** INTEGER)

INIT sets up the symbol table to a maximum depth of MAX\_DEPTH.

**implementation**

Initially we will count the number of variables in the code and create an array of that size. We will simply go through each entry in the array looking for lvalue and label and call PUTS on it to add them into the SYMBOL\_TABLE.

**is composed of** SYNTAX\_VALIDATOR

**end** ANALYZER

**module** SYNTAX\_VALIDATOR

**uses** ERROR\_HANDLING

**exports**

**procedure** VALIDATE (CODE: **in array of** char) **raises** SYNTAX\_ERROR

VALIDATE traverses the code, asserting that the code is syntactically valid.

**implementation**

This module will check certain requirements, such that there is only one call per begin/end block, each begin has a corresponding end, each call or goto (including gofalse and gotrue) has a corresponding label. If any of these requirements are not met, a SYNTAX\_ERROR will be raised.

**end** SYNTAX\_VALIDATOR

**module** STACK\_MANIPULATION

**uses** ERROR\_HANDLING

**exports**

**procedure** push (C: **in** INTEGER)

push will take C, and push it onto its stack.

**procedure** rvalue (L: **in** IDENTIFIER)

ravlue takes the contents at L’s memory address, and pushes it onto the stack.

**procedure** lvalue (L: **in** IDENTIFIER)

lvalue takes the address of L, and pushes it onto the stack.

**procedure** pop **raises** UNDERFLOW\_EXCEPTION

pop simply removes the top element on the stack and throws it away.

**procedure** := **raises** MISSING\_LVALUE\_EXCEPTION

:= takes the value at the top of the stack and assigns it to the address stored in the stack one spot below the top of the stack. Both elements are then popped off of the stack.

**procedure** copy **raises** EMPTY\_STACK\_EXCEPTION

copy takes the value at the top of the stack and pushes a second copy of it onto the stack.

**implementation**

We are assuming only integer values can be pushed to the stack. Nothing too crazy about implementing this. If pop results in an underflow, the ERROR\_HANDLING module will address the error.

**end** STACK\_MANIPULATION

**module** ERROR\_HANDLING

**exports**

**procedure** SYNTAX\_ERROR (TYPE: **in array of** char)

SYNTAX\_ERROR will display an error to the console explaining what went wrong, depending on TYPE.

**procedure** UNDERFLOW\_EXCEPTION

UNDERFLOW\_EXCEPTION occurs when pop is called, but there is no element to pop off of the stack.

**procedure** MISSING\_LVALUE\_EXCEPTION

MISSING\_LVALUE\_EXCEPTION occurs when := is called, but there is no lvalue to assign a value to.

**procedure** EMPTY\_STACK\_EXCEPTION

EMPTY\_STACK\_EXCEPTION occurs when copy is called, but there are no items in the stack to copy.

**implementation**

When an error is raised, the correct procedure is called, a message is displayed to the user, and we exit gracefully.

**end** ERROR\_HANDLING

**module** CONTROL\_FLOW

**uses** STACK\_MANIPULATION, SYMBOL\_TABLE

**exports**

**procedure** label (LABEL\_NAME: **in array of** char)

label is a term used to mark a section of the program. Any **goto**s with the same name as this label will result in the program jumping to this label.

**procedure** goto (LABEL\_NAME: **in array of** char)

goto will result in the program jumping to the line of code that declares a **label** with name LABEL\_NAME.

**procedure** gofalse (LABEL\_NAME: **in array of** char)

gofalse will call **pop** to retrieve the top element on the stack. If the value is 0, the program will jump to the **label** with the name LABEL\_NAME.

**procedure** gotrue (LABEL\_NAME: **in array of** char)

gotrue will call **pop** to retrieve the top element on the stack. If the value is not 0, the program will jump to the **label** with the name LABEL\_NAME.

**procedure** halt

halt will immediately stop program execution.

**implementation**

As execution is processed, if **goto**, **gofalse**, or **gotrue** is found, we will look for a corresponding label with the same LABEL\_NAME as the **go**\* command. If a label is found, we will jump to LABEL\_NAME (if **gofalse** or **gotrue**, the condition must be true to jump).

**end** CONTROL\_FLOW

**module** ARITHMETIC\_OPERATORS

**uses** STACK\_MANIPULATION

**exports**

**procedure** +

+ pops off the top two values, and then pushes the sum of the two values onto the stack.

**procedure** –

- pops off the top two values, and then pushes the difference of the two values onto the stack.

**procedure** \*

\* pops off the top two values, and then pushes the product of the two values onto the stack.

**procedure** /

/ pops off the top two values, and then pushes the quotient of the two values onto the stack.

**procedure** div

div pops off the top two values, and the pushes the remainder of the two values onto the stack.

**end** ARITHMETIC\_OPERATORS

**module** LOGICAL\_OPERATORS

**uses** STACK\_MANIPULATION

**exports**

**procedure** &

& pops off the top two values on the stack, performs a logical AND (not a bitwise AND), and pushes the result onto the stack.

**procedure** !

! pops off the top value on the stack, negates it, and then pushes the result back onto the stack.

**procedure** |

| pops off the top two values on the stack, performs a logical OR (not a bitwise OR), and pushes the result onto the stack.

**end** LOGICAL\_OPERATORS

**module** RELATIONAL\_OPERATORS

**uses** STACK\_MANIPULATION

**exports**

**procedure** <>

<> pops off the top two values on the stack, and returns 0 if the values are equal, or 1 if they are not equal

**procedure** <=

<= pops off the top two values on the stack, and returns 1 if (top – 1) is less than or equal to top; it returns 0 otherwise.

**procedure** >=

>= pops off the top two values on the stack, and returns 1 if (top – 1) is greater than or equal to top; it returns 0 otherwise.

**procedure** <

< pops off the top two values on the stack, and returns 1 if (top – 1) is less than top; it returns 0 otherwise.

**procedure** >

> pops off the top two values on the stack, and returns 1 if (top – 1) is greater than top; it returns 0 otherwise.

**end** RELATIONAL\_OPERATORS

**module** OUTPUT

**uses** STACK\_MANIPULATION

**exports**

**procedure** print

print pops off the top of the stack and outputs its contents to the console.

**procedure** show (OUT: **in** **array of** char)

show immediately outputs the contents of OUTPUT to the console.  
**end** OUTPUT

