**module** MAIN

**uses** CODE\_PARSER

**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. Validation is already handled.

**implementation**

MAIN handles getting and making sure the input file exists and the output filename is valid. It then calls CODE\_PARSER to parse the code into a usable form.

**end** MAIN

**module** CODE\_PARSER

**uses** ANALYZER

**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. It will then call ANALYZER to fill out the SYMBOL\_TABLE.

**end** CODE\_PARSER

**module** ANALYZER

**uses** SYMBOL\_TABLE, ABSTRACT\_TREE\_HANDLER

**exports**

**procedure** ANALYZE (CODE: **in array of** string);

ANALYZE is called to analyze the source and create the SYMBOL\_TABLE. Basically, it will add labels and variables to the SYMBOL\_TABLE.

**implementation**

We will simply go through each entry in the array looking for lvalue, or label and call PUTS on it to add them into the SYMBOL\_TABLE.

**module** STACK\_MANIPULATION

**exports**

**procedure** push (

**procedure** rvalue (

**procedure** lvalue (

**procedure** pop (

**procedure** := (

**procedure** copy (

**end** STACK\_MANIPULATION

**module** CONTROL\_FLOW

**uses** STACK\_MANIPULATION

**exports**

**procedure** label (

**procedure** goto (

**procedure** gofalse (

**procedure** gotrue (

**procedure** halt (

**implementation**

**is composed of**

**end** CONTROL\_FLOW

**module** ARITHMETIC\_OPERATORS

**uses** STACK\_MANIPULATION

**exports**

**procedure** + (

**procedure** - (

**procedure** \* (

**procedure** / (

**procedure** div (

**end** ARITHMETIC\_OPERATORS

**module** LOGICAL\_OPERATORS

**uses** STACK\_MANIPULATION

**exports**

**procedure** & (

**procedure** ! (

**procedure** | (

**end** LOGICAL\_OPERATORS

**module** RELATIONAL\_OPERATORS

**uses** STACK\_MANIPULATION

**exports**

**procedure** <> (

**procedure** <= (

**procedure** >= (

**end** RELATIONAL\_OPERATORS

**module** OUTPUT

**uses** STACK\_MANIPULATION  
**end** OUTPUT