# Design and implementation of a one-pass macro processor

# Project Report

# *CMPE 220 (Spring 2012) – Dr. Weider D. Yu*

***Department of Computer Engineering, San Jose State University***

***May 15, 2012***

**Team 1**

|  |  |
| --- | --- |
| ***Andrew H. Lam***  ***SID: 006519825***  ***Email:*** [***andrew.h.lam@gmail.com***](mailto:andrew.h.lam@gmail.com) | ***Jonathan Guan***  ***SID: 008091915***  ***Email:*** [***jonguan@gmail.com***](mailto:jonguan@gmail.com) |
| ***Mujtaba M. Hassanpur***  ***SID: 008159346***  ***Email:*** [***mhassanpur@gmail.com***](mailto:mhassanpur@gmail.com) | ***Rajesh Somasundaran***  ***SID: 008064563***  ***Email:*** [***rsomasundaran@gmail.com***](mailto:rsomasundaran@gmail.com) |

# Introduction

# This group project is intended to demonstrate our understanding of a one-pass macro processor. We have implemented a macro processor for SIC/XE assembly programs. The algorithm used for the macro processor implementation was taken from the textbook “System Software – An Introduction to Systems Programming,” by Leland L. Beck.

# Architecture and Design

# Design analysis

We have taken the approach of a simple yet modular software design in our implementation of the macro processor. Major functionalities of the macro processor viz. DEFINE, GETLINE, PROCESSLINE, EXPAND, etc. are isolated into separate modules. Interfaces between the modules are established by means of clearly defined function parameters.

The modular approach allows the whole implementation or parts of it to be reused in conjunction with other system software such as assemblers. This approach also provides greater readability of the code, and also helps in better maintainability and reusability. For example, one decision that we made early in the coding was to create a parsing data structure *parse\_info\_t* which would copy input from a buffer, error check the line for format, and then divide the line into the label, opcode, and operands. This allowed us to quickly and efficiently operate on the data within each module without having to recreate the parsing process. We also decided to keep a pointer to a buffer containing the current line, so that each module could communicate to the other modules as to where to operate next.

The design was made with extensibility capabilities in mind, so that the basic functionality could be implemented first and then extension of it with additional features could be done easily. We first implemented NAMTAB, ARGTAB, and DEFTAB as described in Fig 4.4 in the book. However, in order to support extended features, as well as to optimize resource utilization and access speed, we rewrote ARGTAB using a hash table. Conditional Macro Expansion was made possible by upgrading the EXPAND module.

The low level design also accounted for a performance efficient and secure implementation. Inside functions, memory is allocated only when needed and are freed when no longer needed. Secure versions of library functions are used wherever possible, and file operations (such as open/close, read/write) are done in such a way that the I/O time is reduced.

# Flow Charts

### Macro-processor Main Procedure

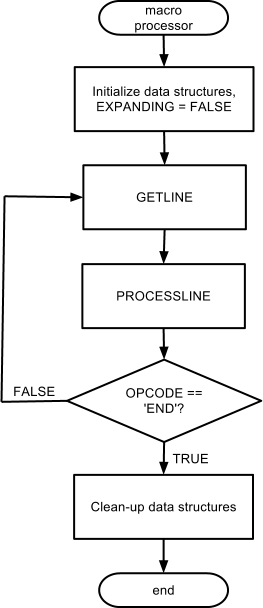


Figure 3.1 - Main Procedure Flowchart

### PROCESSLINE Procedure

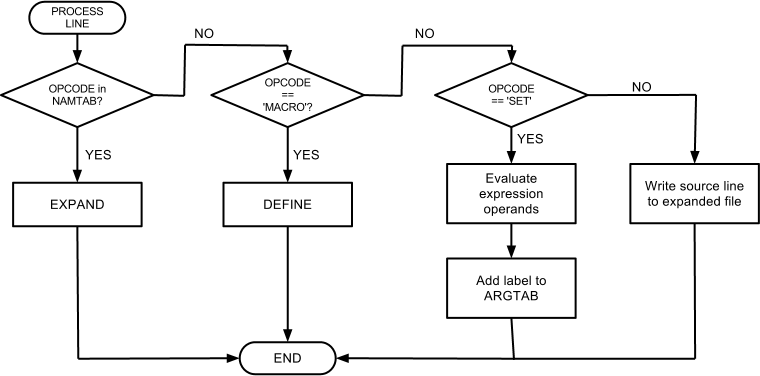


FIgure 3.2 - PROCESSLINE Procedure Flowchart

### DEFINE Procedure

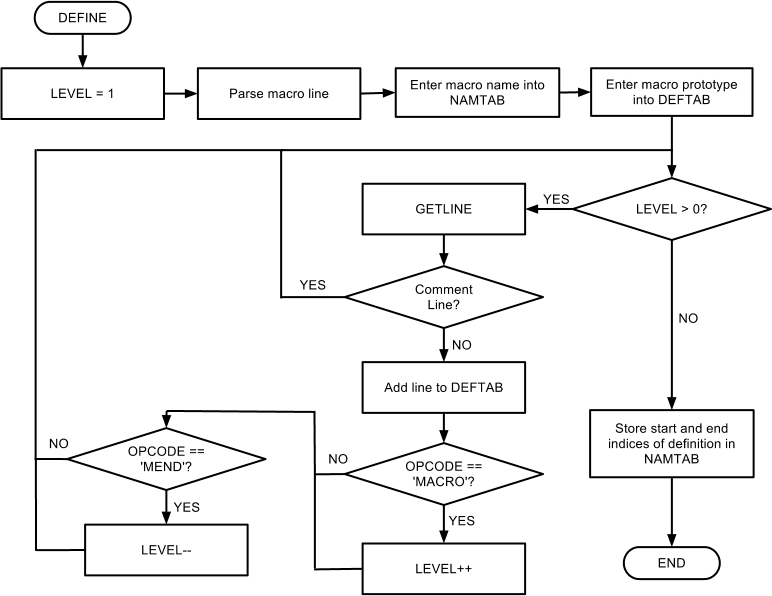


Figure 3.3 - DEFINE Procedure Flowchart

### EXPAND Procedure

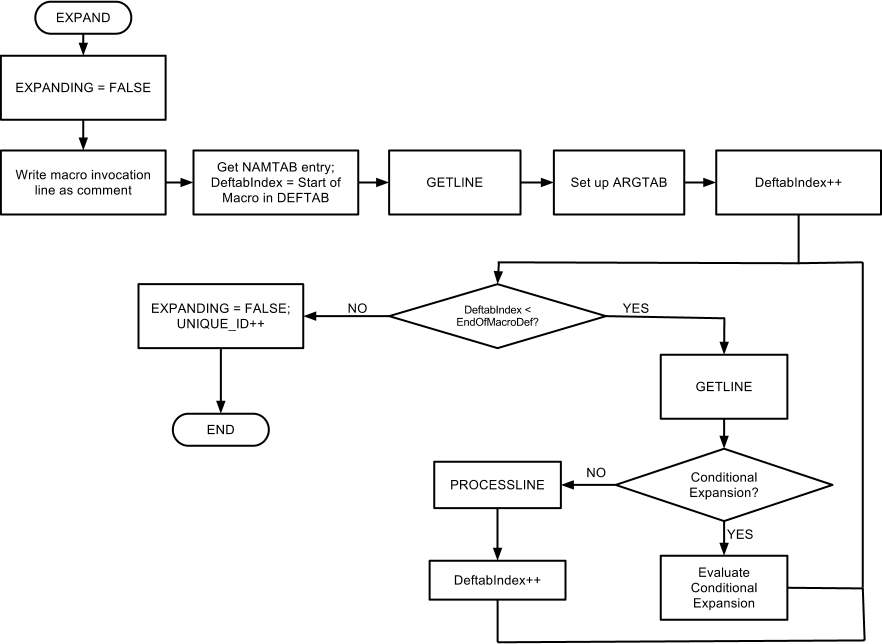


Figure 3.4 - EXPAND Procedure Flowchart

### GETLINE Procedure

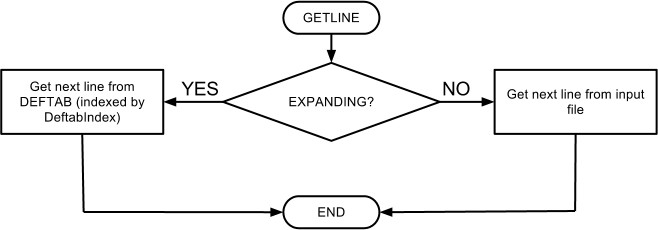


Figure 3.5 - GETLINE Procedure Flowchart

1. **Algorithms and Data Structures**

# Algorithm

The algorithm described in Figure 4.5 of the prescribed textbook [1] presents an algorithm for a one-pass macro processor. It worked well for converting Fig 4.1 source code into Fig 4.2 output. However in order to implement Keyword Macro Expansion and Conditional Macro Expansion, and nested forms of IF/WHILE, we had to extend the algorithm and modify the ARGTAB. It presents an algorithm for a one-pass macro processor.

* + 1. **Modifications to EXPAND for Conditional Macro Expansion**

In order for Conditional Macro Expansion to work, we needed a parser to evaluate the check inside the IF or WHILE statements. The original parse\_line method would break up the source input into 3 sections per line: label, opcode, and operands. Conditional Macro Expansion worked very nicely with the parse\_line method, as the opcode would be IF or WHILE, and the condition to evaluate would be in operands. In cases of SET as the opcode, the label would contain the variable to set, and the operands would contain the result on which to set the variable.

In order to evaluate the conditional expressions however, we needed a mathematical and symbolic evaluator. The evaluator would need to substitute variables inside the mathematical expressions, and then mathematically evaluate the result. In cases of %NITEMS, the parser would call a sub-function to iterate and parse the expression. The parser would also need to replace expressions using array values, which relied on modifications to the ARGTAB. Because this parser is a non-trivial task worthy of its own project, we decided to only handle simple mathematical functions of +, -, \*, /, and %. It does not handle parens, exponents, polynomial expansion, nor any form of calculus.

Finally, for nested conditional expansions, we needed a data structure to keep track of the level of nesting, as well as to keep track of whether to evaluate the sections as we iterated through the DEFTAB lines. This was implemented with a simple array of integers, functioning like a stack. For each IF/ELSE/ENDIF and WHILE/ENDW clause, we stored the result of the evaluation of the IF clause in the array. If evaluation of WHILE/ENDW clauses were still valid, the DEFTAB line number of the WHILE clause was stored in the results array. Finally, we needed an easy way to skip sections where the conditional expansions evaluated to false. For example, in an IF/ELSE/ENDIF clause, where the IF section contains additional nested WHILE and/or IF/ELSE/ENDIF sections, we would want to skip the entire parent IF clause if the condition evaluated to false. This is implemented through storing an integer denoting the action to skip all lines until the nested conditions finish through the opcodes ENDIF and ENDW.

* + 1. **Modifications to ARGTAB and DEFINE for Conditional Macro Expansion and Keyword Macro Expansion**

In Figure 4.4, the ARGTAB is implemented as a simple array. However, in order to support default parameter values for Keyword Macro Expansion, as well as for null inputs for Conditional Macro Expansion, a simple array would no longer work. This is because in the C language, arrays are terminated with null. There are ways to avoid this problem, such as storing a ‘ ‘ or other symbol to denote null; having a count of arguments to expect; appending to the array in cases of SET; iterating the array for each while loop to find the variable to replace, etc.

However, the more elegant solution is to replace the array structure with a hash table. Using a hash table makes variable replacement a O(1) constant time operation, rather than a O(n) operation. This is greatly valuable, especially for while loops and long parameter lists.

The change to Fig 4.5 is that instead of replacing the parameters with ?1, ?2, ?3…?n positional notation during the DEFINE function, we would simply input the definition line into the DEFTAB. During EXPAND, the definition line is read, matched with the invocation line, and the variables are set during EXPAND. Moreover, as the macro processor encounters macro-specific variables, those are easily added to the hash table in ARGTAB as well.

Finally, because of %NITEMS and array value replacement, the ARGTAB needed extra functionality to keep track of which items were arrays and which items were simple tokens. This was accomplished by adding a Boolean to the hash table struct node. This helped the parser to easily identify times to select a value within an array versus replacing the variable with the entire array.

* 1. **Data Structures**

Data structures are implemented in the same format as mentioned in the textbook. We used dynamically growing arrays to implement many of the data structures. This allowed for flexible, and memory efficient use of the data structures.

* + 1. **ARGTAB (includes UTHASH, an open source software from sourceforge.net)**

A hash table is used for the ARGTAB. It adds or sets symbols and its corresponding value to the ARGTAB. This made it easier to look up the symbols directly. Figure 3.6 shows a logical view of the ARGTAB data structure. The following function prototypes, defined in argtab.h and implemented in argtab.c, describe the ARGTAB API:

**argtab\_t \*** argtab\_alloc(**void**);

**void** argtab\_free(**argtab\_t \*** table);

**int** argtab\_add(**argtab\_t \*** table, **const** **char** **\*** symbol, **const** **char** **\*** value);

**char** **\*** argtab\_get(**argtab\_t \*** table, **const** **char** **\*** symbol);

**int** argtab\_set(**argtab\_t \*** table, **const** **char** **\*** symbol, **const** **char** **\*** value);

**int** argtab\_addOrSet(**argtab\_t \*** table, **const** **char** **\*** symbol, **const** **char** **\*** value);

**void** argtab\_clear(**argtab\_t \*** table);

**void** argtab\_substituteValues(**argtab\_t \*** table, **char** **\*** buffer, **size\_t** bufsize);

One function of note is argtab\_substituteValues, which would replace parameter names with the argument values passed in through the invocation. During Conditional Macro Expansion, this function would also handle array indexing such as &EOR[2].

* + 1. **DEFTAB**

Dynamically growing arrays were used for the DEFTAB to store the strings. It was essentially an array of strings. Figure 3.7 shows a logical view of the DEFTAB data structure. The following function prototypes, defined in deftab.h and implemented in deftab.c, describe the DEFTAB API:

**deftab\_t \*** deftab\_alloc(**void**);

**void** deftab\_free(**deftab\_t \***);

**int** deftab\_add(**deftab\_t \*** table, **const** **char** **\*** data);

**char** **\*** deftab\_get(**deftab\_t \*** table, **int** index);

* + 1. **NAMTAB**

Dynamically growing arrays were also used for the NAMTAB to store entries that are associated with each symbol. Figure 3.8 shows a logical view of the NAMTAB data structure. The following function prototypes, defined in namtab.h and implemented in namtab.c, describe the NAMTAB API:

**namtab\_t \*** namtab\_alloc(**void**);

**void** namtab\_free(**namtab\_t \*** table);

**int** namtab\_add(**namtab\_t \*** table, **const** **char** **\*** symbol, **int** start, **int** end);

**namtab\_entry\_t \*** namtab\_get(**namtab\_t \*** table, **const** **char** **\*** symbol);

**namtab\_entry\_t \*** namtab\_getIndex(**namtab\_t \*** table, **int** index);

* + 1. **Parse\_info\_t**

The parsing function did not include growing arrays, but did include extra metadata about each line. After parsing each line, the struct would contain information on whether the line was a comment line (line starts with a period), on whether the line contained keyword macro parameters, as well as the different parts of the line (label, opcode, and operands). This structure allows the macroprocessor to reconstruct the line in a standard predefined format, even if the input is non-standard (ie. opcode starts at different columns).

typedef struct

{

BOOL isComment;

BOOL hasKeywordMacroParameters;

char \* label;

char \* opcode;

char \* operators;

} parse\_info\_t;

**parse\_info\_t \*** parse\_info\_alloc(**void**);

**void** parse\_info\_free(**parse\_info\_t \*** parse\_info);

**void** parse\_info\_clear(**parse\_info\_t \*** parse\_info);

**void** parse\_info\_print(**parse\_info\_t \*** parse\_info);

**int** parse\_line(**parse\_info\_t \*** parse\_info, **const char \*** line);

**int** parse\_reconstruct\_string(**parse\_info\_t \*** parse\_info, **char \***returnString);

### ARGTAB Data Structure

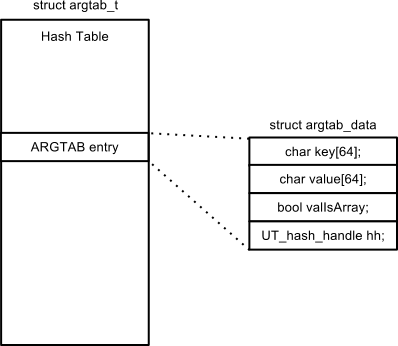


Figure 3.6 - ARGTAB Data Structure

### DEFTAB Data Structure

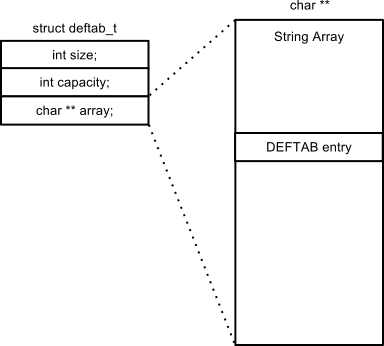


Figure 3.7 - DEFTAB Data Structure

### NAMTAB Data Structure

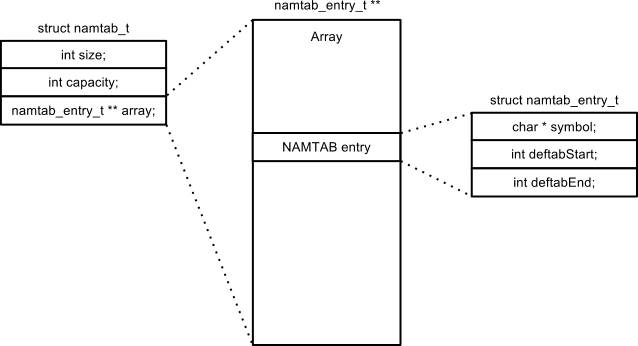


Figure 3.8 - NAMTAB Data Structure

1. **Implementation Details**
   1. **Software Development Lifecycle**

We followed the *Iterative and Incremental* Software Development Life Cycle (SDLC). We started with basic functionality based on the algorithm in Figure 4.5 of the Book [1] and then added additional features after achieving basic functionality.

* 1. **Tools used:**
* Visual Studio 2010 (IDE)
* Github.com (Code Repository & revision control system)

The tools used in the development of this project helped to manage and iterate the software design cycle quickly and efficiently. Each member used Visual Studio, which helped to fix bugs we discovered them. However, we derived most of our success from installing and using github, an online source repository. This allowed us to develop independently on Visual Studio and then compare, commit, and push our changes online. Github also allowed us to comment on the code, file bugs, and prevent code conflicts.

We highly recommend code controls for any team collaborating on software.

## Open Source References:

* UTHash (Hashtable for C structures; from <http://uthash.sourceforge.net>)

Special thanks to Troy D. Hanson for his work on UTHash. It is implemented as a lightweight linked list with hash functions, easily imported into our project.

1. **Features Implemented**
2. **Unique Labels Generation**: When implementing this feature, we used a new integer to keep track of the number of macro invocations. Each invocation thus had a unique identifier. In encountering a unique label, we simply converted the unique identifier (an integer) to its corresponding two-letter representation (i.e. ‘AA’).
3. **Keyword Macro Parameters**: For this feature, we converted our initial ARGTAB implementation to one that used a hash table (UT Hash), to enable faster look up of symbols and their corresponding values. In addition, we removed the use of positional notation in DEFTAB entries and their subsequent substitution in GETLINE. Instead, we stored parameter names directly into DEFTAB as they appeared in the source code. We then used the ARGTAB values to substitute keys (definition parameters) with their corresponding values (invocation parameters). This also paved the way for the Conditional Macro Expansion feature.
4. **Conditional Macro Expansion**: Checks for one of the conditional expansion keywords. It parses the line and evaluates the conditional statement. This also handles nested conditional statements (nested IF/WHILE within another IF/WHILE) similar to nested macros. This feature also adds great flexibility to use default values for parameters when the argument in the invocation is null.
5. **Mathematical and Symbolic Evaluator**:Used to evaluate conditional Macro Expansion, this evaluator would also be used for macro specific functions, such as %NITEMS and SET. Its job is to accurately parse the expression, replace variables found, and then evaluate the resulting expression.
6. **Dynamic Parsing and Pretty Printing of Output File**: Even if the input is non-standard and non-uniform, the macro processor is capable of printing the output in a standard pre-defined format. For example, if the opcode for each line do not always start at column 17, the macro processor would still be able to recognize it and after processing, would print on column 17. This is critical for communication with other system software, such as assemblers, which may rely on a fixed parsing algorithm.
7. **Testing**

Our implementation was tested thoroughly to ensure quality of the software. Each module was subjected to unit testing, followed by integration testing. Bug fixes were unit tested first and then subjected to regression testing to ensure quality and stability of the code while development was in progress.

1. **Conclusion**

As explained under section 2.1, we followed a strong design for the development of this software. All stages of a software development life cycle were followed to ensure faster development of this software with quality. The incremental approach helped us to implement a basic functioning macro processor first and then to add more features as time permitted. Bug fixes were done quickly and were tested thoroughly. Each module was subjected to unit testing, followed by integration testing. A strong revision control solution was implemented from the beginning of the project that enabled greater level of collaboration in this software development.

The project checkpoints earlier in the semester helped us to plan and architect the project early, rather than haphazardly dividing work and diving in. The project checkpoint also allowed us to partition work, decide on extra features, and responsibilities for implementing them. Even though the end result did not perfectly match the earlier blueprint, the sheer act of planning the work early saved us plenty of headache later.

While implementing the additional features not described in Fig 4.5, we realized that we needed to modify the project. We had to decide on new data structures, which functions to extend, which functions to expand, and which functions to replace. For example, the decision on whether to implement SET in processLine, expand, or getline led to a discussion on the core identity of each function. (We decided processLine, true to its name, should process each line, and therefore process SET as well.) Another example is the replacement of the ARGTAB array structure with a hash table structure in order to kill 3 birds with 1 stone: Keyword Macro Expansion, Default Parameter Values, and fast variable replacement.

We added numerous error-checking lines throughout the code, which helped us debug our macro processor when it was not working as expected.

This project not only helped us to gain in-depth knowledge of system software but also the end-to-end software development process. The team project also helped us to gain soft skills of working together as a team.

1. **Project Contribution**
2. Jon: MAIN, PROCESSLINE, Conditional Macro Expansion, Macro processor Functions (%NITEMS, SET, array indexing), Pretty printing of output file.
3. Andrew: GETLINE, Test Cases, Documentation
4. Mujtaba: DEFINE. Unique labels generation, Keyword Macro Parameters, Data Structures (ARGTAB, DEFTAB, NAMTAB, PARSER)
5. Rajesh: EXPAND, Test Cases, Documentation

# Attachments:

1. Source Code
2. Test Cases
3. Input Files, and Output Files (Results)
4. README
5. Project Report (this document)
6. **References**
   1. System Software – An Introduction to Systems Programming: Leland L. Beck: 3rd Edition
   2. [www.github.com](http://www.github.com)
   3. MSDN Library and API Reference: <http://msdn.microsoft.com/en-us/library/ms123401.aspx>

**Appendix A: Test Cases**

Test Case #1

Run the program with file Fig4-1.txt

In RDBUFF, =X'&INDEV' should become =X'F1'

Test Case #2

Run the program with file Fig4-1.txt

In RDBUFF, &BUFADR,X should become BUFFER,X

Test Case #3

Run the program with file Fig4-1.txt

In WRBUFF, =X'&OUTDEV' should become =X'05'

Test Case #4

Run the program with file Fig4-8.txt

In RDBUFF, EOR character =X'&EOR' should become =X'04'

Test Case #5

Run the program with file Fig4-8.txt

In RDBUFF, &RECLTH should become LENGTH

Test Case #6

Run the program with file Fig4-8.txt

There should be 3 start loops: $AALOOP, $ABLOOP, and $ACLOOP

Test Case #7

Run the program with file Fig4-8.txt

There should be 3 exit loops: $AAEXIT, $ABEXIT and $ACEXIT

Test Case #8

Run the program with file Fig4-8.txt

#&MAXLTH should become #80

Test Case #9

Run the program with file Fig4-9.txt

This WHILE loop will evaluate (&CTR LE &EORCT) and this =X'0000&EOR[&CTR]' should become =X'000000', =X'000003', and =X'000004'

Test Case #10

Run the program with file Fig4-9.txt

&RECLTH should become LENGTH

Test Case #11

Run the program with file SimpleIfWhile.txt

The first line to print out is "FIRST NE BLANK"

Test Case #12

Run the program with file SimpleIfWhile.txt

The program will loop 10 times and display "HOORAY" 10 times.

Test Case #13

Run the program with file SimpleIfWhile.txt

The program will loop 10 times and display "HIP" 5 times and "HIP2" 5 times.

Test Case #14

Run the program with file TestKeyWord.txt

=X'&OUTDEV' should become =X'05'

Test Case #15

Run the program with file TestKeyWord.txt

=X'&INDEV' should become =X'F1'

Test Case #16

Run the program with file TestUniqueLabels.txt

There should be 4 start loops: $AALOOP, $ABLOOP, $AELOOP, and $AFLOOP

Test Case #17

Run the program with file TestUniqueLabels.txt

There should be 4 exit loops: $AAEXIT, $ABEXIT, $AEEXIT, and $AFEXIT