TomPiler

Generated by Doxygen 1.9.3

Chapter 1

TomPiler

Version

0.2.5

1.0.1 Useful Pages

- compfiles.h
- file_util.h
- dfa.h
- tokens.h
- scan.h
- TCompFiles
- Scanner

1.0.2 About

Created by Group 3 for CSC-460, Language Translations with Dr. Pyzdrowski, at PennWest California.

Chapter 2

changelog

3/20/2023: Karl, Thomas

- Wrote the final printing stuff for compilation result

3/20/2023: Karl

- ParseError region
- Appropriate handling of parse and lex errors
- Pretty printing of errors
- Limited parse error recovery
- Full lex error recovery
- Error summarys

3/19/2023: Karl, Thomas, Anthony

- Debugged lexical error printing

3/19/2023: Karl, Anthony

- Debugged parser
- Debugged SkipWhitespaceforNextToken in scanner
- Printing statements for parser
- Init and lifecycle stuff for parser in tompiler

3/19/2023: Karl

- Wrote parse functions

3/18/2023: Karl, Thomas

- First few parse statements, including Parse_Program, Parse_SystemGoal, StatementList, Statement
- Ensured some printing to listing file and output file were occurring
- Fixed comment issue with skipping whitespace

3/11/2023: Karl

- Added rubric, examples to /doc
- Renamed terminal.h to console.h because terminal has a different meaning in the context of parsing.
- Added CONSOLE_COLOR and CONSOLE_COLOR_DEFAULT macros in console.h
- Added Scanner_LoadFiles(input,output,listing,temp) as part of scanner lifecycle methods. *note* professor may want us to pass in the input file at each call... I prefer to do this the object oriented way but I'll want to check to make sure he won't dock us for that.
- Added Scanner_CopyBuffer(char * destination) which copies the contents of scanner's buffer to another chararray and adds a null terminator.
- Added Scanner_SkipAllWhitespaceForNextToken() which also skips newlines, which is used for Scanner_↩ NextToken checks (because the next token may be on a new line)
- Added Scanner_NextToken() which gets the next token in the input file, then returns the file pointer to its original position (before all whitespace)
- Removed leftover dfa test that was confirming minusop read as minusop when adjacent to intliteral; we changed this just before submission to read the way he wanted (as a negative intliteral) but I never removed the test.
- Added scanner_test.c which will be used to validate Scanner_NextToken and Scanner_Match. Utility functions for creating temp input files and initing scanner. Wrote tests to validate LookAhead and Match.

2/16/2023: Karl and Thomas

- Changed DFA to result in error on strings like '99a'; identifiers must start with characters, can't start with numbers.
- Changed DFA to not automatically process negative numbers. Rationale is that we want to allow cases like 100-100 to be read as INTLITERAL MINUSOP INTLITERAL not INTLITERAL INTLITERAL. Therefore, negatizing the intliterals is a context-sensitive activity that will occur at parse time instead of scan time.
- Reverted the intliteral and identifier changes

2/15/2023: Karl and Thomas

- Better printing for compfiles opening and closing and better print formatting.
- Negative intliterals on dfa.
- Tompiler.h
- Brought back the expanding buffer! Using it for actual token strings now instead of an entire line.
- Brought in terminal.h for virtual terminal color sequences and ported it to C.
- Pretty printing for Hello and Goodbye.

2/15/2023: Karl

- Finished debugging full DFA.
- Rewrote scanner in scan.h. No more using memory allocation and TokenCatch structures. It reads from file and writes directly to the listing and output files now.
- Deleted scanner.c, scanner.h, scanner_util.c, scanner_util.h and associated test files.
- Deleted the recognize keyword token function and the associated dfa, since the new dfa covers everything.
- If we need those files and features back, we can revert to an earlier commit.

2/14/2023: Karl

- Created a FULL dfa planned to replace all current logic.
- Moved all test header files to one header file "test.h".

2/13/2023: Karl and thomas

- fixed listing file not loading
- fixed extractInt AND extractInt tests (they were using extractWord)!
- running error count and print errors
- fixed detect SCANEOF
- fixed null terminates at end of buffer for no overflow print
- fixed last line being ignored
- several types of token catch initializers
- token catch allocates memory; can use the tokens later in the parser
- scanner printLine fixes
- print error count
- Token_GetOpRaw
- formatting line printing

2/13/2023: All Group Members

- Extract op
- Fleshed out the switch statement for take Action
- printLine always happens at the end of populateBuffer now
- extractOperator and Scanner_ExtractOperator.... extractOperator is in Scanner not scanner_util because it is dependent
- Token_CatchOp Token_CatchError

2/12/2023: Karl

- Added recognizers for trueop, nullop, falseop to the state transition table, which I had missed before.

- Skipwhitespace now returns the number of characters missed. This can be useful if we extract a number and it isn't followed by a whitespace (skipwhitespace will produce 0.) This may be a cause for an error print. (Worth asking)
- Fixed extractWord errors and added Scanner_ExtractWord
- Added extractInteger and Scanner_ExtractInteger
- Added a boundries member to Tscanner. This is a list of all boundry characters that delimit words, identifiers, and number and it includes all operators plus whitespace and EOF. See Scanner_Init() for how it's constructed.
- Token Recognize now returns ERROR if there is a non number, non alphanumeric within the tested string. It also now allows for identifiers to have numbers.
- Created Token wrapping struct called TokenCatch that encapsulates info about the token such as the recognized raw string, line number, and so forth.
- Scanner now takes files on Scanner_Scan(files...) not on Scanner_Init()
- Init and DeInit functions in main
- Moved switch statement/dispatcher into a function Scanner_TakeAction(lookaheadResult)
- Made basic Scanner_Scan(); currently will print the listing file numbers and lines only

2/11/2023: Karl

- Added skipWhitespace general function in scanner_util and added tests for it
- Added charIn function in scanner_util which is used by extractWord function in scanner_util.

2/10/2023: Karl

- Added Scanner_populateBuffer() and tested it.
- Created Scanner_LookAhead() and put a switch statement in Scanner_Scan()

2/8/2023: Karl, Thomas, Anthony

- created scanner.h and scanner.c
- add struct to hold scanner info
- scanner lifecycle functions
- scanner buffer functions
- scanner_util , created buffer resize and refresh functions
- created tests for scanner_util

2/7/2023: Karl

- abstracted command line argument parse and calls to a new function, CompFiles_FileOpenFromCLIArgs, which also generates the Temp file.
- created Tokens_GetName, the tokens Map, the tokens enum.

- created a state transition table as a 3d array for a keyword recognizer Token_RecognizeKeyword. Used excel to design the table; .xlsx is in the /docs folder.

2/1/2023 : All Group Members

- used fileapi.h to create a getAbsolutePath function
- created checkIfSamePaths function to compare file name actual paths
- reworked the validate files functions to check for output/input name collisions
- adjusted some printing
- fixed tempfile bug

1/28/2023: Karl

- used doxygen to generate documentation

1/27/2023: Thomas, Karl

- wrote copy inputs to outputs function

1/26/2023 : All group members

- refactored file_util into two files: compfiles and file_util
- worked on logic for validating an output file name
- auto-generate temp file
- validate listing file in a similar way to output file
- combined validation functions into one validate func; just pass it the command line arguments

1/25/2023 : Thomas

- promptOutputFile()
- Modified getString() to use realloc

1/24/2023 : All group members

- worked on main logic
- changed CompFiles struct to be a state machine
- created promptFilename

1/23/2023 : Thomas and klm127

- changed Author comment to include e-mail and class name.
- removed old addExtension function, old promptFilename function, and closeFile function.
- added promptFilename and getString function(not yet covered by unit tests)
- removed all of the stdin swapping to a `separate repo`, and tested it, due to nagging bugs.
  - NOTE: It turned out that the bug was that `dup2` closes a file and `fclose` was being called afterwards.

moved test dependencies to a sub folder `lib` and updated compilation commands to use this on the include path
1/22/2023 : thomas and klm127

- added removeExtension function and tests
- confirmed getchar will read an 'enter'.
- thomas fixed prompting function to accept alternate inputs
- added backupFile function and tests

- Included tests for filepaths with directories
- redid filenameHasExtension. It now allows for filenames like ".bob" and doesnt allow filenames that end in slashes. It does allow folders to have '.'s in them.

1/21/2023 : klm127

- added #pragma region directives to header files. This is basically just markup for VSCode. Each of these regions can now be folded in Visual Studio or VSCode. This does not affect -ansi compilation on MinGW-W64 gcc; as far as I can tell. The purpose is to make the code much easier to navigate without relying on tab-based folding. `See Also: stackoverflow answer`
- Cleaned up comments, tab-based folding, etc.
- Fixed up the `addExtension` to use malloc to create a longer, concatenated string out of its inputs. Added unit tests for `addExtension`.
- Refactored std swapping test utility functions. The best way to test a prompter is now to use is to call `setSTDin3`, get the value, then dont forget to call `restoreSTD3()` *before* making a test-based assertion.

1/20/2023 : All group members in collaboration

- created `promptUserOverwriteSelection`.
- created tests for `promptUserOverwriteSelection`. This was quite an involved task because we had to figure out how to temporarily replace stdin and stdout with alternative files so that we could test functionalities like `scanf`. Ultimately we were able to figure it out.

1/19/2023 : klm127

- changed directory structure, added docs, src, and tests
- created changelog, included CuTest's readme in the docs
- updated tasks.json in .vscode to configure code generation
- output file is now `fileopen.exe` due to interpretation of video instructions
- added .gitignore so we can exclude executables from github
- Added the testing suite CuTest. More info `here`
- Added the functions `fileExists` and `filenameHasExtension`
- Added unit tests for `fileExists` and `filenameHasExtension`

Chapter 3

VSCode setup instructions

VSCode provides a decent environment to work in C with its highly customizable features, low overhead, and rich extension options.
The folder `.vscode` configures the workspace for use with VSCode.
`tasks.json` describes build and run commands.
Ctrl+Shift+B will build and run the programs.
You may have to change compilerPath in c_cpp_properties.json to your own compiler.
I'm using GCC 8.1 (came with CodeBlocks) with the -ansi flag.
I referenced this article when setting up the VSCode environment. `Medium Article`
I referenced the `gcc documentation` while setting up the compiler.

Chapter 4

Tompiler Readme

Tompiler will be a relatively simple compiler built for educational and explorative purposes.

4.1 Compiling

Compiler configurations are stored in the .bat files. There are two of them.

- runTests.bat compiles and runs the tests.
- compile.bat compiles and runs the code.

4.2 Using

Running `compile.bat` will run the compiler after executing. You can also find the executable, `fileopen.exe`, in your `bin` directory.
It takes up to two command line arguments. The first argument can be an input file path while the second argument can be an output file path.
Place the `bin` directory on your system path if you want to be able to run tompiler from anywhere.

4.3 Folder and file Descriptions

- .vscode : Contains vscode configurations.
- docs : Contains additional documentation
- src : Contains source code
  - main.c : Program entry point
  - compfiles.c / .h : struct for managing input output file access
  - file_util.c / .h : file i/o helpersfor the compiler
  - dfa.c / .h : The DFA which drives the scanning process.
  -
- tests : Contains source code for tests
  - lib: Contains test dependencies
    - CuTest.c / .h : CuTest micro test framework
    - std_swapper.c / .h : For swapping stdin and out with files.
  - file_util_test.c : tests for file util
  - dfa_test.c : tests for dfa.
  - tokens_test.c : test for token functions.
  - main_test.c : entry point for test compilation
  - tests.h : each test file has one exported member, a function that returns the testing suite. They are all declared here.

4.4 Included 3rd party library, CuTest.

Link to Cutest page
This is a small bit of code (only 340 lines!) that provides a unit testing skeleton.

4.5 Credits

- Tom Terhune
- Karl Miller
- Anthony Stepich

Chapter 5

Deprecated List

Global Scanner_ScanAndPrint (FILE *input, FILE *listing, FILE *output, FILE *temp)

Chapter 6

Todo List

Global Scanner_Match (int target_token)

Checks for newlines and prints to the listing file the line if one is found.

Advances column position.

Chapter 7

Data Structure Index

7.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 8

File Index

8.1 File List

Here is a list of all files with brief descriptions:

Chapter 9

Data Structure Documentation

9.1 Scanner Struct Reference

`#include <scan.h>`

Data Fields

- int line_no
- int col_no
- int errors
- char * buffer
- int capacity
- int l_buffer
- FILE * in
- FILE * out
- FILE * temp
- FILE * listing

9.1.1 Detailed Description

Scanner struct holds references to the files being read and keeps track of the line and column position. It is a singleton.

9.1.2 Field Documentation

9.1.2.1 buffer

`char* buffer`

9.1.2.2 capacity

`int capacity`

9.1.2.3 col_no

`int col_no`
The column number.

9.1.2.4 errors

```
int errors
```
The error count.

9.1.2.5 in

```
FILE* in
```

9.1.2.6 l_buffer

```
int l_buffer
```

9.1.2.7 line_no

```
int line_no
```
The line number being scanned.

9.1.2.8 listing

```
FILE* listing
```

9.1.2.9 out

```
FILE* out
```

9.1.2.10 temp

```
FILE* temp
```
The documentation for this struct was generated from the following file:

- src/scan.h

9.2 T_Parser Struct Reference

```
#include <parse.h>
```

Data Fields

- FILE * out
- FILE * list
- char * buffer
- int capacity
- int l_buffer
- int errorCount
- int trace

9.2.1 Field Documentation

9.2.1.1 buffer

```
char* buffer
```
A buffer, for printing completed statements.

9.2.1.2 capacity

`int capacity`
The current capacity of the buffer

9.2.1.3 errorCount

`int errorCount`
A tally of total syntax errors.

9.2.1.4 l_buffer

`int l_buffer`
The length of relevant characters in the buffer, and the write index.

9.2.1.5 list

`FILE* list`
The listing file

9.2.1.6 out

`FILE* out`
The output file

9.2.1.7 trace

`int trace`
A tally of how many functions have failed, for printing them nestedly.
The documentation for this struct was generated from the following file:

- src/parse.h

9.3 TCompFiles Struct Reference

Manages input and output files.
`#include <compfiles.h>`

Data Fields

- FILE * in
- FILE * out
- FILE * temp
- FILE * listing
- short input_file_state
- short output_file_state
- short listing_file_state
- short terminate_requested
- short has_requested_default_filename
- char * input_file_name
- char * output_file_name
- char * listing_file_name
- char * temp_file_name

9.3.1 Detailed Description

Manages input and output files.
CompFiles is a globally accesible struct which maintains references to the loaded files.
It has a number of functions closely associated to it. In that way it is a class-like, but a singleton. There is only one CompFiles that ever should exist.

9.3.2 Field Documentation

9.3.2.1 has_requested_default_filename

`short has_requested_default_filename`
1 indicates that a user has requested to use a default ouput filename already. This is so that if the user selects this twice, they will automatically exit instead of looping the prompt.

9.3.2.2 in

`FILE* in`
A file pointer to an open input file.

9.3.2.3 input_file_name

`char* input_file_name`
The input filename.

9.3.2.4 input_file_state

`short input_file_state`
Determines the status of input file validation.

9.3.2.5 listing

`FILE* listing`
A file pointer to an open listing file.

9.3.2.6 listing_file_name

`char* listing_file_name`
The listing filename

9.3.2.7 listing_file_state

`short listing_file_state`
Determines the status of listing file validation.

9.3.2.8 out

`FILE* out`
A file pointer to an open output file.

9.3.2.9 output_file_name

`char* output_file_name`
The output filename,

9.3.2.10 output_file_state

`short output_file_state`
Determines the status of output file validation.

9.3.2.11 temp

`FILE* temp`
A file pointer to an open tmp file.

9.3.2.12 temp_file_name

```
char* temp_file_name
```

The temp filename

9.3.2.13 terminate_requested

```
short terminate_requested
```

1 indicates that a user requested to terminate the program.
The documentation for this struct was generated from the following file:

- src/compfiles.h

9.4 TokenCatch Struct Reference

```
#include <tokens.h>
```

Data Fields

- short token
- char * raw
- int line_no
- int col_no

9.4.1 Detailed Description

9.4.1.1 Note: TokenCatch is no longer used. It was used in an earlier version of this program. It may be revived in the future depending on the needs of the parser.

9.4.2 Field Documentation

9.4.2.1 col_no

```
int col_no
```

9.4.2.2 line_no

```
int line_no
```

9.4.2.3 raw

```
char* raw
```

9.4.2.4 token

```
short token
```

The documentation for this struct was generated from the following file:

- src/tokens.h

Chapter 10

File Documentation

10.1 docs/changelog.md File Reference

10.2 docs/VSCode.md File Reference

10.3 Readme.md File Reference

10.4 src/compfiles.c File Reference

CompFiles struct and "methods" definitions.
`#include "compfiles.h"`

Functions

- void CompFiles_Init ()
- void CompFiles_GenerateTempFile ()
- void CompFiles_DeInit ()
- TCompFiles * CompFiles_GetFiles ()
- void CompFiles_LoadInputFile (FILE *newInputFile)
- void CompFiles_LoadOutputFile (FILE *newOutputFile)
- void CompFiles_LoadTempFile (FILE *newTempFile)
- void CompFiles_LoadListingFile (FILE *newListingFile)
- char * CompFiles_promptInputFilename ()
- void CompFiles_CopyInputToOutputs ()
- void CompFiles_AppendTempToOut ()
- short CompFiles_Open (int argc, char *argv[ ])
- short CompFiles_AcquireValidatedFiles (char *inputFilename, const char *outputFilename)
- short CompFiles_AcquireValidatedInputFile (char *filename)
- short CompFiles_AcquireValidatedOutputFile (const char *filename)
- short CompFiles_AcquireValidatedListingFile (const char *filename)
- char * CompFiles_promptOutputFilename ()
- short CompFiles_promptUserOverwriteSelection ()

10.4.1 Detailed Description

CompFiles struct and "methods" definitions.
CompFiles is a struct which holds pointers to the compilation input and output files. It also tracks their names and their validation status. It provides methods for prompting the user for valid file names until terminate is requested or all files are validated.

Authors

Tom Terhune, Karl Miller, Anthony Stepich

Date

January 2023

10.4.2 Function Documentation

10.4.2.1 CompFiles_AcquireValidatedFiles()

```
short CompFiles_AcquireValidatedFiles (
            char * inputFilename,
            const char * outputFilename )
```

Loops and prompts until all input and output files are set correctly or until terminate is requested. After the input, output, and listing files are generated, CompFiles_AcquireValidatedFiles also generates a temp file.

Parameters

inputFilename	a filename with which to begin input validation with or NULL
outputFilename	a filename with which to begin output validation with or NULL

Returns

1 if terminate was requested. Otherwise, 0.

Author

klm127

Date

1/26/2023

10.4.2.2 CompFiles_AcquireValidatedInputFile()

```
short CompFiles_AcquireValidatedInputFile (
            char * filename )
```

Validates an input file name and sets the value in the struct. It will continue looping until the user has supplied a valid filename or elected to quit the program.

Parameters

filename	a filename with which to begin input validation with or NULL

Returns

0 if the input file was validated and loaded into the struct. 1 if the user requested to terminate the program.

10.4.2.3 CompFiles_AcquireValidatedListingFile()

```
short CompFiles_AcquireValidatedListingFile (
            const char * filename )
```

Validates a listing file name and sets the value in the struct.
Called by CompFiles_ValidateOutputFile after an output file has been fully validated. The parameter passed will be the name of the output file with the extension 'list' instead.
If this file happens to exist, a similar loop will occur as when a user attempts to load an extant output file. The user will be prompted to enter a new file until one is validated or they elect to exit the program.

Parameters

filename	a filename with which to begin input validation with or NULL

Returns

0 if an output file was validated and loaded into the struct. 1 if the user requested to terminate the program.

10.4.2.4 CompFiles_AcquireValidatedOutputFile()

```
short CompFiles_AcquireValidatedOutputFile (
            const char * filename )
```

Validates an output file name and sets the value in the struct. It will continue looping until the user has supplied a valid filename or elected to quit the program.

Parameters

filename	a filename with which to begin input validation with or NULL

Returns

0 if an output file was validated and loaded into the struct. 1 if the user requested to terminate the program.

10.4.2.5 CompFiles_AppendTempToOut()

```
void CompFiles_AppendTempToOut ( )
```

append the temp file to the out file

10.4.2.6 CompFiles_CopyInputToOutputs()

```
void CompFiles_CopyInputToOutputs ( )
```

CompFiles_CopyInputToOutputs copies all the data from the input file to each of the output files. After execution, all output files (tmp, list, and out) will have text identical to the input files.

Warning

Precondition: All CompFiles file pointers must be open and ready to read/write.

Author

Thomas, Karl

Date

1/27/2023

10.4.2.7 CompFiles_DeInit()

```
void CompFiles_DeInit ( )
```

Closes any open files and returns CompFiles to the default values. Deletes the temp file.

10.4.2.8 CompFiles_GenerateTempFile()

```
void CompFiles_GenerateTempFile ( )
```

Generates a temporary file with a unique name. This file will be destroyed when CompFiles_DeInit() is called.

Author

klm127

Date

1/26/2023

10.4.2.9 CompFiles_GetFiles()

```
TCompFiles * CompFiles_GetFiles ( )
```

Gets the CompFiles struct so that the validated files can be used elsewhere in the program.

Returns

A TCompFiles struct.

10.4.2.10 CompFiles_Init()

```
void CompFiles_Init ( )
```

Initializes CompFiles struct to default values.

Note

Covered by unit tests.

10.4.2.11 CompFiles_LoadInputFile()

```
void CompFiles_LoadInputFile (
            FILE * newInputFile )
```

CompFiles_LoadInputFile loads a new file pointer as the input file. If there is a file already loaded, it closes that file first.

Parameters

newInputFile	A pointer to an open file in read mode.

10.4.2.12 CompFiles_LoadListingFile()

```
void CompFiles_LoadListingFile (
            FILE * newListingFile )
```

CompFiles_LoadListingFile loads a new file pointer as the listing file. If there is a file already loaded, it closes that file first.

Parameters

newOutputFile	A pointer to an open file in write mode.

10.4.2.13 CompFiles_LoadOutputFile()

```
void CompFiles_LoadOutputFile (
            FILE * newOutputFile )
```

CompFiles_LoadOutputFile loads a new file pointer as the output file. If there is a file already loaded, it closes that file first.

Parameters

newOutputFile	A pointer to an open file in write mode.

10.4.2.14 CompFiles_LoadTempFile()

```
void CompFiles_LoadTempFile (
            FILE * newTempFile )
```

CompFiles_LoadTempFile loads a new file pointer as the temp file. If there is a file already loaded, it closes that file first.

Parameters

newOutputFile	A pointer to an open file in write mode.

10.4.2.15 CompFiles_Open()

```
short CompFiles_Open (
            int argc,
            char * argv[] )
```

Parses the command line args and calls functions to acquire validated filenames.

Parameters

argc	The argument count.
argv	The argument array.

Returns

1 if terminate was requested. Otherwise, 0.

Author

klm127

Date

2/7/2023

10.4.2.16 CompFiles_promptInputFilename()

```
char * CompFiles_promptInputFilename ( )
```

Calls the function getString() to recieve a filename from the user and returns it. It will set the 'terminate requested' flag in CompFiles if the user inputs only a \n.

Returns

char * inputfilename to be verified

Author

thomaserh99

Date

1/23/2023

Note

Covered by Unit Tests

10.4.2.17 CompFiles_promptOutputFilename()

```
char * CompFiles_promptOutputFilename ( )
```

Calls the function getString() to recieve a filename from the user and returns it. It will set the 'terminate requested' flag in CompFiles if the user inputs only a \n.

Warning

This should not be called until the input filename has been set. The user may elect to generate an output filename based on the input file. (inputfilename + .out)

Returns

A malloced string of an output filename to be verified.

Author

thomaserh99

Date

Created On: 1/23/2023

Note

Covered by Unit Tests

10.4.2.18 CompFiles_promptUserOverwriteSelection()

```
short CompFiles_promptUserOverwriteSelection ( )
```

Prompts the user as to what they want to do about an output file already existing. It prints a prompt and parses the user response to one of the USER_OUTPUT_OVERWRITE_SELECTION enums. It does NOT loop.

Returns

short corresponding to one of the enums of USER_OTUPUT_OVERWRITE_SELECTION

Author

klm127, thomasterh99, anthony91501

Date

1/20/2023

Note

Covered by Unit Tests

10.5 src/compfiles.h File Reference

CompFiles struct and "methods".

```
#include <stdio.h>
#include "file_util.h"
#include <string.h>
#include <stdlib.h>
```

Data Structures

- struct TCompFiles

  Manages input and output files.

Enumerations

- enum COMPFILES_STATE { COMPFILES_STATE_NO_NAME_PROVIDED = 0 , COMPFILES_STATE_NAME_NEEDS_VALID = 1 , COMPFILES_STATE_NAME_VALIDATED = 2 }
- enum USER_OUTPUT_OVERWRITE_SELECTION { USER_OUTPUT_OVERWRITE_REENTER_FILENAME_SELECTED = 1 , USER_OUTPUT_OVERWRITE_OVERWRITE_EXI = 2 , USER_OUTPUT_OVERWRITE_DEFAULT_FILENAME = 3 , USER_OUTPUT_TERMINATE_PROGRAM = 4 , USER_OUTPUT_TERMINATE_INVALID_ENTRY = -1 }

Functions

- void CompFiles_Init ()
- void CompFiles_DeInit ()
- void CompFiles_GenerateTempFile ()
- TCompFiles * CompFiles_GetFiles ()
- void CompFiles_LoadInputFile (FILE *newInputFile)
- void CompFiles_LoadOutputFile (FILE *newOutputFile)
- void CompFiles_LoadTempFile (FILE *newTempFile)
- void CompFiles_LoadListingFile (FILE *newListingFile)
- short CompFiles_Open (int argc, char *argv[ ])
- short CompFiles_AcquireValidatedFiles (char *inputFilename, const char *outputFilename)
- short CompFiles_AcquireValidatedInputFile (char *filename)
- short CompFiles_AcquireValidatedOutputFile (const char *filename)
- short CompFiles_AcquireValidatedListingFile (const char *filename)
- char * CompFiles_promptInputFilename ()
- char * CompFiles_promptOutputFilename ()
- short CompFiles_promptUserOverwriteSelection ()
- void CompFiles_CopyInputToOutputs ()
- void CompFiles_AppendTempToOut ()

Variables

- TCompFiles CompFiles

10.5.1 Detailed Description

CompFiles struct and "methods".
CompFiles is a struct which holds pointers to the compilation input and output files. It also tracks their names and their validation status. It provides methods for prompting the user for valid file names until terminate is requested or all files are validated.

Authors

Tom Terhune, Karl Miller, Anthony Stepich

Date

January 2023

10.5.2 Enumeration Type Documentation

10.5.2.1 COMPFILES_STATE

`enum COMPFILES_STATE`
Describes the state of a filename validation process

Enumerator

COMPFILES_STATE_NO_NAME_PROVIDED	
COMPFILES_STATE_NAME_NEEDS_VALIDATION	
COMPFILES_STATE_NAME_VALIDATED	

10.5.2.2 USER_OUTPUT_OVERWRITE_SELECTION

`enum USER_OUTPUT_OVERWRITE_SELECTION`
Describes the possible selections a user may make when they elect to output to a file that already exists.

Enumerator

USER_OUTPUT_OVERWRITE_REENTER_FILENAME_SELECTED	
USER_OUTPUT_OVERWRITE_OVERWRITE_EXISTING_FILE	
USER_OUTPUT_OVERWRITE_DEFAULT_FILENAME	
USER_OUTPUT_TERMINATE_PROGRAM	
USER_OUTPUT_TERMINATE_INVALID_ENTRY	

10.5.3 Function Documentation

10.5.3.1 CompFiles_AcquireValidatedFiles()

```
short CompFiles_AcquireValidatedFiles (
            char * inputFilename,
            const char * outputFilename )
```

Loops and prompts until all input and output files are set correctly or until terminate is requested. After the input, output, and listing files are generated, CompFiles_AcquireValidatedFiles also generates a temp file.

Parameters

inputFilename	a filename with which to begin input validation with or NULL
outputFilename	a filename with which to begin output validation with or NULL

Returns

1 if terminate was requested. Otherwise, 0.

Author

klm127

Date

1/26/2023

10.5.3.2 CompFiles_AcquireValidatedInputFile()

```
short CompFiles_AcquireValidatedInputFile (
            char * filename )
```

Validates an input file name and sets the value in the struct. It will continue looping until the user has supplied a valid filename or elected to quit the program.

Parameters

filename	a filename with which to begin input validation with or NULL

Returns

0 if the input file was validated and loaded into the struct. 1 if the user requested to terminate the program.

10.5.3.3 CompFiles_AcquireValidatedListingFile()

```
short CompFiles_AcquireValidatedListingFile (
            const char * filename )
```

Validates a listing file name and sets the value in the struct.
Called by CompFiles_ValidateOutputFile after an output file has been fully validated. The parameter passed will be the name of the output file with the extension 'list' instead.
If this file happens to exist, a similar loop will occur as when a user attempts to load an extant output file. The user will be prompted to enter a new file until one is validated or they elect to exit the program.

Parameters

filename	a filename with which to begin input validation with or NULL

Returns

0 if an output file was validated and loaded into the struct. 1 if the user requested to terminate the program.

10.5.3.4 CompFiles_AcquireValidatedOutputFile()

```
short CompFiles_AcquireValidatedOutputFile (
            const char * filename )
```

Validates an output file name and sets the value in the struct. It will continue looping until the user has supplied a valid filename or elected to quit the program.

Parameters

filename	a filename with which to begin input validation with or NULL

Returns

0 if an output file was validated and loaded into the struct. 1 if the user requested to terminate the program.

10.5.3.5 CompFiles_AppendTempToOut()

```
void CompFiles_AppendTempToOut ( )
```

append the temp file to the out file

10.5.3.6 CompFiles_CopyInputToOutputs()

```
void CompFiles_CopyInputToOutputs ( )
```

CompFiles_CopyInputToOutputs copies all the data from the input file to each of the output files. After execution, all output files (tmp, list, and out) will have text identical to the input files.

Warning

Precondition: All CompFiles file pointers must be open and ready to read/write.

Author

Thomas, Karl

Date

1/27/2023

10.5.3.7 CompFiles_DeInit()

```
void CompFiles_DeInit ( )
```

Closes any open files and returns CompFiles to the default values. Deletes the temp file.

10.5.3.8 CompFiles_GenerateTempFile()

```
void CompFiles_GenerateTempFile ( )
```

Generates a temporary file with a unique name. This file will be destroyed when CompFiles_DeInit() is called.

Author

klm127

Date

1/26/2023

10.5.3.9 CompFiles_GetFiles()

```
TCompFiles * CompFiles_GetFiles ( )
```

Gets the CompFiles struct so that the validated files can be used elsewhere in the program.

Returns

A TCompFiles struct.

10.5.3.10 CompFiles_Init()

```
void CompFiles_Init ( )
```

Initializes CompFiles struct to default values.

Note

Covered by unit tests.

10.5.3.11 CompFiles_LoadInputFile()

```
void CompFiles_LoadInputFile (
            FILE * newInputFile )
```

CompFiles_LoadInputFile loads a new file pointer as the input file. If there is a file already loaded, it closes that file first.

Parameters

newInputFile	A pointer to an open file in read mode.

10.5.3.12 CompFiles_LoadListingFile()

```
void CompFiles_LoadListingFile (
            FILE * newListingFile )
```

CompFiles_LoadListingFile loads a new file pointer as the listing file. If there is a file already loaded, it closes that file first.

Parameters

newOutputFile	A pointer to an open file in write mode.

10.5.3.13 CompFiles_LoadOutputFile()

```
void CompFiles_LoadOutputFile (
            FILE * newOutputFile )
```

CompFiles_LoadOutputFile loads a new file pointer as the output file. If there is a file already loaded, it closes that file first.

Parameters

newOutputFile	A pointer to an open file in write mode.

10.5.3.14 CompFiles_LoadTempFile()

```
void CompFiles_LoadTempFile (
            FILE * newTempFile )
```

CompFiles_LoadTempFile loads a new file pointer as the temp file. If there is a file already loaded, it closes that file first.

Parameters

newOutputFile	A pointer to an open file in write mode.

10.5.3.15 CompFiles_Open()

```
short CompFiles_Open (
            int argc,
            char * argv[] )
```

Parses the command line args and calls functions to acquire validated filenames.

Parameters

argc	The argument count.
argv	The argument array.

Returns

1 if terminate was requested. Otherwise, 0.

Author

klm127

Date

2/7/2023

10.5.3.16 CompFiles_promptInputFilename()

```
char * CompFiles_promptInputFilename ( )
```

Calls the function getString() to recieve a filename from the user and returns it. It will set the 'terminate requested' flag in CompFiles if the user inputs only a \n.

Returns

char * inputfilename to be verified

Author

thomaserh99

Date

1/23/2023

Note

Covered by Unit Tests

10.5.3.17 CompFiles_promptOutputFilename()

```
char * CompFiles_promptOutputFilename ( )
```

Calls the function getString() to recieve a filename from the user and returns it. It will set the 'terminate requested' flag in CompFiles if the user inputs only a \n.

Warning

This should not be called until the input filename has been set. The user may elect to generate an output filename based on the input file. (inputfilename + .out)

Returns

A malloced string of an output filename to be verified.

Author

thomaserh99

Date

Created On: 1/23/2023

Note

Covered by Unit Tests

10.5.3.18 CompFiles_promptUserOverwriteSelection()

```
short CompFiles_promptUserOverwriteSelection ( )
```

Prompts the user as to what they want to do about an output file already existing. It prints a prompt and parses the user response to one of the USER_OUTPUT_OVERWRITE_SELECTION enums. It does NOT loop.

Returns

short corresponding to one of the enums of USER_OTUPUT_OVERWRITE_SELECTION

Author

klm127, thomasterh99, anthony91501

Date

1/20/2023

Note

Covered by Unit Tests

10.5.4 Variable Documentation

10.5.4.1 CompFiles

```
TCompFiles CompFiles
```

The CompFiles singleton.

10.6 compfiles.h

Go to the documentation of this file.

```
1
2 #ifndef compfiles_h
3 #define compfiles_h
4
16 #include <stdio.h>
17 #include "file_util.h"
18 #include <string.h>
19 #include <stdlib.h>
20
21 /*
22 -----------------
23 CompFiles typedef
24 -----------------
25 */
26 #pragma region structs
28 enum COMPFILES_STATE {
29     COMPFILES_STATE_NO_NAME_PROVIDED = 0,
30     COMPFILES_STATE_NAME_NEEDS_VALIDATION = 1,
31     COMPFILES_STATE_NAME_VALIDATED = 2
32 };
33
41 typedef struct {
43     FILE * in;
45     FILE * out;
47     FILE * temp;
49     FILE * listing;
51     short input_file_state;
53     short output_file_state;
55     short listing_file_state;
57     short terminate_requested;
59     short has_requested_default_filename;
61     char * input_file_name;
63     char * output_file_name;
65     char * listing_file_name;
67     char * temp_file_name;
68 } TCompFiles;
69
71 TCompFiles CompFiles;
72
73 #pragma endregion structs
74
75 /*
76 -------------------
77 CompFiles lifecycle
78 -------------------
79 */
80 #pragma region lifecycle
81
85 void CompFiles_Init();
89 void CompFiles_DeInit();
96 void CompFiles_GenerateTempFile();
97
102 TCompFiles* CompFiles_GetFiles();
103
104 #pragma endregion lifecycle
105
106 /*
107 -----------------
108 CompFiles setters
109 -----------------
110 */
111 #pragma region setters
112
116 void CompFiles_LoadInputFile(FILE * newInputFile);
117
121 void CompFiles_LoadOutputFile(FILE * newOutputFile);
122
126 void CompFiles_LoadTempFile(FILE * newTempFile);
127
131 void CompFiles_LoadListingFile(FILE * newListingFile);
132
133 #pragma endregion setters
134
135 /*
136 ------------------
137 CompFiles prompts
138 ------------------
139 */
140 #pragma region prompts
141
142
152 short CompFiles_Open(int argc, char *argv[]);
153
```

```
154
164 short CompFiles_AcquireValidatedFiles(char * inputFilename, const char * outputFilename);
165
166
173 short CompFiles_AcquireValidatedInputFile(char * filename);
174
181 short CompFiles_AcquireValidatedOutputFile(const char * filename);
182
193 short CompFiles_AcquireValidatedListingFile(const char * filename);
194
204 char * CompFiles_promptInputFilename();
205
217 char * CompFiles_promptOutputFilename();
218
222 enum USER_OUTPUT_OVERWRITE_SELECTION {
223     USER_OUTPUT_OVERWRITE_REENTER_FILENAME_SELECTED = 1,
224     USER_OUTPUT_OVERWRITE_OVERWRITE_EXISTING_FILE = 2,
225     USER_OUTPUT_OVERWRITE_DEFAULT_FILENAME = 3,
226     USER_OUTPUT_TERMINATE_PROGRAM = 4,
227     USER_OUTPUT_TERMINATE_INVALID_ENTRY = -1
228 };
240 short CompFiles_promptUserOverwriteSelection();
241
242 #pragma endregion prompts
243
244 /*
245 --------------------
246 CompFiles operations
247 --------------------
248 */
249 #pragma region operations
250
260 void CompFiles_CopyInputToOutputs();
261
262
263
268 void CompFiles_AppendTempToOut();
269
270 #pragma endregion operations
271
272
273 #endif
274
```

10.7 src/console.h File Reference

Windows Console Macros.

Macros

- #define ESC "\x1B["
- #define CSI "\x1B"
- #define GRAPHIC "m"
- #define UNDERLINE 4
- #define NO_UNDERLINE 24
- #define FG_BLACK "\x1b[30m"
- #define FG_DEFAULT "\x1b[39m"
- #define FG_YELLOW "\x1b[33m"
- #define FG_MAGENTA "\x1b[35m"
- #define FG_CYAN "\x1b[36m"
- #define FG_WHITE "\x1b[37m"
- #define FG_BLUE "\x1b[34m"
- #define FG_RED L"x1b[31m"
- #define FG_GREEN "\x1b[32m"
- #define FG_BRT_RED "\x1b[91m"
- #define FG_BRT_GREEN "\x1b[92m"
- #define FG_BRT_BLACK "\x1b[90m"
- #define FG_BRT_YELLOW "\x1b[93m"
- #define FG_BRT_BLUE "\x1b[94m"
- #define FG_BRT_GREEN "\x1b[92m"

- #define FG_BRT_MAGENTA "\x1b[95m"
- #define FG_BRT_CYAN "\x1b[96m"
- #define FG_BRT_WHITE "\x1b[97m"
- #define BG_DEFAULT "\x1b[49m"
- #define BG_BLUE "\x1b[44m"
- #define BG_MAGENTA "\x1b[45m"
- #define BG_RED L"x1b[41m"
- #define BG_GREEN "\x1b[42m"
- #define BG_BLACK "\x1b[40m"
- #define BG_WHITE "\x1b[47m"
- #define BG_YELLOW "\x1b[43m"
- #define BG_BRT_RED "\x1b[101m"
- #define BG_BRT_BLACK "\x1b[100m"
- #define BG_BRT_GREEN "\x1b[102m"
- #define BG_BRT_YELLOW "\x1b[103m"
- #define BG_BRT_BLUE "\x1b[104m"
- #define BG_BRT_MAGENTA "\x1b[105m"
- #define BG_BRT_CYAN "\x1b[106m"
- #define BG_BRT_WHITE "\x1b[107m"
- #define CONSOLE_COLOR(FG, BG) printf("%s%s", FG,BG)
- #define CONSOLE_COLOR_DEFAULT() printf("%s%s", FG_DEFAULT, BG_DEFAULT)

10.7.1 Detailed Description

Windows Console Macros.
Provides macros for using Console Virtual Terminal Sequences
Enable Virtual Terminal Sequences flag must be set in Console Mode to use.
See: https://docs.microsoft.com/en-us/windows/console/console-virtual-terminal-sequence
See: https://learn.microsoft.com/en-us/windows/console/getconsolemode

Author

Karl Miller

10.7.2 Macro Definition Documentation

10.7.2.1 BG_BLACK

```
#define BG_BLACK "\x1b[40m"
```

10.7.2.2 BG_BLUE

```
#define BG_BLUE "\x1b[44m"
```

10.7.2.3 BG_BRT_BLACK

```
#define BG_BRT_BLACK "\x1b[100m"
```

10.7.2.4 BG_BRT_BLUE

```
#define BG_BRT_BLUE "\x1b[104m"
```

10.7.2.5 BG_BRT_CYAN

```
#define BG_BRT_CYAN "\x1b[106m"
```

10.7.2.6 BG_BRT_GREEN

```
#define BG_BRT_GREEN "\x1b[102m"
```

10.7.2.7 BG_BRT_MAGENTA

```
#define BG_BRT_MAGENTA "\x1b[105m"
```

10.7.2.8 BG_BRT_RED

```
#define BG_BRT_RED "\x1b[101m"
```

10.7.2.9 BG_BRT_WHITE

```
#define BG_BRT_WHITE "\x1b[107m"
```

10.7.2.10 BG_BRT_YELLOW

```
#define BG_BRT_YELLOW "\x1b[103m"
```

10.7.2.11 BG_DEFAULT

```
#define BG_DEFAULT "\x1b[49m"
```

10.7.2.12 BG_GREEN

```
#define BG_GREEN "\x1b[42m"
```

10.7.2.13 BG_MAGENTA

```
#define BG_MAGENTA "\x1b[45m"
```

10.7.2.14 BG_RED

```
#define BG_RED L"x1b[41m"
```

10.7.2.15 BG_WHITE

```
#define BG_WHITE "\x1b[47m"
```

10.7.2.16 BG_YELLOW

```
#define BG_YELLOW "\x1b[43m"
```

10.7.2.17 CONSOLE_COLOR

```
#define CONSOLE_COLOR(
             FG,
             BG ) printf("%s%s", FG,BG)
```

10.7.2.18 CONSOLE_COLOR_DEFAULT

```
#define CONSOLE_COLOR_DEFAULT( ) printf("%s%s", FG_DEFAULT, BG_DEFAULT)
```

10.7.2.19 CSI

```
#define CSI "\x1B"
```

10.7.2.20 ESC

```
#define ESC "\x1B["
```

10.7.2.21 FG_BLACK

```
#define FG_BLACK "\x1b[30m"
```

10.7.2.22 FG_BLUE

```
#define FG_BLUE "\x1b[34m"
```

10.7.2.23 FG_BRT_BLACK

```
#define FG_BRT_BLACK "\x1b[90m"
```

10.7.2.24 FG_BRT_BLUE

```
#define FG_BRT_BLUE "\x1b[94m"
```

10.7.2.25 FG_BRT_CYAN

```
#define FG_BRT_CYAN "\x1b[96m"
```

10.7.2.26 FG_BRT_GREEN [1/2]

```
#define FG_BRT_GREEN "\x1b[92m"
```

10.7.2.27 FG_BRT_GREEN [2/2]

```
#define FG_BRT_GREEN "\x1b[92m"
```

10.7.2.28 FG_BRT_MAGENTA

```
#define FG_BRT_MAGENTA "\x1b[95m"
```

10.7.2.29 FG_BRT_RED

```
#define FG_BRT_RED "\x1b[91m"
```

10.7.2.30 FG_BRT_WHITE

```
#define FG_BRT_WHITE "\x1b[97m"
```

10.7.2.31 FG_BRT_YELLOW

```
#define FG_BRT_YELLOW "\x1b[93m"
```

10.7.2.32 FG_CYAN

```
#define FG_CYAN "\x1b[36m"
```

10.7.2.33 FG_DEFAULT

```
#define FG_DEFAULT "\x1b[39m"
```

10.7.2.34 FG_GREEN

```
#define FG_GREEN "\x1b[32m"
```

10.7.2.35 FG_MAGENTA

```
#define FG_MAGENTA "\x1b[35m"
```

10.7.2.36 FG_RED

```
#define FG_RED L"x1b[31m"
```

10.7.2.37 FG_WHITE

```
#define FG_WHITE "\x1b[37m"
```

10.7.2.38 FG_YELLOW

```
#define FG_YELLOW "\x1b[33m"
```

10.7.2.39 GRAPHIC

```
#define GRAPHIC "m"
```

10.7.2.40 NO_UNDERLINE

```
#define NO_UNDERLINE 24
```

10.7.2.41 UNDERLINE

```
#define UNDERLINE 4
```

10.8 console.h

Go to the documentation of this file.

```
1 #ifndef k_terminal
2 #define k_terminal
3
20 /* Terminal escape sequences */
21
22 #define ESC "\x1B["
23 #define CSI "\x1B"
24
25 #define GRAPHIC "m"
26
27 #define UNDERLINE 4
28 #define NO_UNDERLINE 24
29
30 #define FG_BLACK "\x1b[30m"
31 #define FG_DEFAULT "\x1b[39m"
32 #define FG_YELLOW "\x1b[33m"
33 #define FG_MAGENTA "\x1b[35m"
34 #define FG_CYAN "\x1b[36m"
35 #define FG_WHITE "\x1b[37m"
36 #define FG_BLUE "\x1b[34m"
37 #define FG_RED L"x1b[31m"
38 #define FG_GREEN "\x1b[32m"
39 #define FG_BRT_RED "\x1b[91m"
40 #define FG_BRT_GREEN "\x1b[92m"
41 #define FG_BRT_BLACK "\x1b[90m"
42 #define FG_BRT_YELLOW "\x1b[93m"
43 #define FG_BRT_BLUE "\x1b[94m"
44 #define FG_BRT_GREEN "\x1b[92m"
45 #define FG_BRT_MAGENTA "\x1b[95m"
46 #define FG_BRT_CYAN "\x1b[96m"
47 #define FG_BRT_WHITE "\x1b[97m"
48 #define BG_DEFAULT "\x1b[49m"
49 #define BG_BLUE "\x1b[44m"
50 #define BG_MAGENTA "\x1b[45m"
51 #define BG_RED L"x1b[41m"
52 #define BG_GREEN "\x1b[42m"
53 #define BG_BLACK "\x1b[40m"
54 #define BG_WHITE "\x1b[47m"
55 #define BG_YELLOW "\x1b[43m"
56 #define BG_BRT_RED "\x1b[101m"
57 #define BG_BRT_BLACK "\x1b[100m"
58 #define BG_BRT_GREEN "\x1b[102m"
59 #define BG_BRT_YELLOW "\x1b[103m"
60 #define BG_BRT_BLUE "\x1b[104m"
61 #define BG_BRT_MAGENTA "\x1b[105m"
62 #define BG_BRT_CYAN "\x1b[106m"
63 #define BG_BRT_WHITE "\x1b[107m"
64
65 #define CONSOLE_COLOR(FG, BG) printf("%s%s", FG,BG)
66 #define CONSOLE_COLOR_DEFAULT() printf("%s%s", FG_DEFAULT, BG_DEFAULT)
67
68 #endif
69
70
```

10.9 src/dfa.c File Reference

The DFA and related logic definitions.

```
#include "tokens.h"
#include "dfa.h"
#include <string.h>
#include <stdio.h>
```

Enumerations

- enum DFA_STATES {
  STATE_START , STATE_ID , STATE_ERROR , STATE_B ,

STATE_BE , STATE_BEG , STATE_BEGI , STATE_BEGIN ,
STATE_E , STATE_EN , STATE_END , STATE_R ,
STATE_RE , STATE_REA , STATE_READ , STATE_I ,
STATE_IF , STATE_T , STATE_TH , STATE_THE ,
STATE_THEN , STATE_EL , STATE_ELS , STATE_ELSE ,
STATE_ENDI , STATE_ENDIF , STATE_ENDW , STATE_ENDWH ,
STATE_ENDWHI , STATE_ENDWHIL , STATE_ENDWHILE , STATE_W ,
STATE_WH , STATE_WHI , STATE_WHIL , STATE_WHILE ,
STATE_F , STATE_FA , STATE_FAL , STATE_FALS ,
STATE_FALSE , STATE_TR , STATE_TRU , STATE_TRUE ,
STATE_N , STATE_NU , STATE_NUL , STATE_NULL ,
STATE_LPAR , STATE_RPAR , STATE_SEMIC , STATE_COMMA ,
STATE_COLON , STATE_COLONEQUALS , STATE_PLUS , STATE_MINUS ,
STATE_MULTIPLY , STATE_DIV , STATE_NOT , STATE_LESS ,
STATE_LESSEQ , STATE_GREAT , STATE_GREATEQ , STATE_EQ ,
STATE_NOTEQ , STATE_INT , STATE_EOF , STATE_WR ,
STATE_WRI , STATE_WRIT , STATE_WRITE }

- enum DFA_CHARS {
  CH_A , CH_B , CH_C , CH_D ,
  CH_E , CH_F , CH_G , CH_H ,
  CH_I , CH_J , CH_K , CH_L ,
  CH_M , CH_N , CH_O , CH_P ,
  CH_Q , CH_R , CH_S , CH_T ,
  CH_U , CH_V , CH_W , CH_X ,
  CH_Y , CH_Z , CH_WSPC , CH_LPRN ,
  CH_RPRN , CH_SEMIC , CH_COMM , CH_COLON ,
  CH_EQU , CH_PLUS , CH_MINUS , CH_STAR ,
  CH_DIV , CH_NOT , CH_LT , CH_GT ,
  CH_NUM , CH_EOF , CH_NOTINSET , CH_NLINE }

Functions

- char * GetStateString (int n)
- char * GetDFAColString (int n)
- short GetDFAColumn (char c)
- int GetNextToken (FILE *file, int *charsRead)
- int GetNextTokenInBuffer (char *buffer, int *bufIndex, int *charsRead)
- void printCell (int row, int col)
- void printStateAndChar (int row, int col)

Variables

- short DFA [71][44][3]

10.9.1 Detailed Description

The DFA and related logic definitions.
The DFA is a 3 dimensional array that maps a given state and character input to a result consisting of the next state, token, and whether reading should continue.
The DFA was created in Excel, and the excel file is available in docs/fullDFA.xlsx.

Authors

Tom Terhune, Karl Miller, Anthony Stepich

Date

February 2023

10.9.2 Enumeration Type Documentation

10.9.2.1 DFA_CHARS

enum DFA_CHARS

Enumerator

CH_A	
CH_B	
CH_C	
CH_D	
CH_E	
CH_F	
CH_G	
CH_H	
CH_I	
CH_J	
CH_K	
CH_L	
CH_M	
CH_N	
CH_O	
CH_P	
CH_Q	
CH_R	
CH_S	
CH_T	
CH_U	
CH_V	
CH_W	
CH_X	
CH_Y	
CH_Z	
CH_WSPC	
CH_LPRN	
CH_RPRN	
CH_SEMIC	
CH_COMM	
CH_COLON	
CH_EQU	
CH_PLUS	
CH_MINUS	
CH_STAR	
CH_DIV	
CH_NOT	
CH_LT	
CH_GT	
CH_NUM	
CH_EOF	
CH_NOTINSET	

Enumerator

CH_NLINE	

10.9.2.2 DFA_STATES

`enum DFA_STATES`

Enumerator

STATE_START	
STATE_ID	
STATE_ERROR	
STATE_B	
STATE_BE	
STATE_BEG	
STATE_BEGI	
STATE_BEGIN	
STATE_E	
STATE_EN	
STATE_END	
STATE_R	
STATE_RE	
STATE_REA	
STATE_READ	
STATE_I	
STATE_IF	
STATE_T	
STATE_TH	
STATE_THE	
STATE_THEN	
STATE_EL	
STATE_ELS	
STATE_ELSE	
STATE_ENDI	
STATE_ENDIF	
STATE_ENDW	
STATE_ENDWH	
STATE_ENDWHI	
STATE_ENDWHIL	
STATE_ENDWHILE	
STATE_W	
STATE_WH	
STATE_WHI	
STATE_WHIL	
STATE_WHILE	
STATE_F	
STATE_FA	
STATE_FAL	
STATE_FALS	
STATE_FALSE	

Enumerator

STATE_TR	
STATE_TRU	
STATE_TRUE	
STATE_N	
STATE_NU	
STATE_NUL	
STATE_NULL	
STATE_LPAR	
STATE_RPAR	
STATE_SEMIC	
STATE_COMMA	
STATE_COLON	
STATE_COLONEQUALS	
STATE_PLUS	
STATE_MINUS	
STATE_MULTIPLY	
STATE_DIV	
STATE_NOT	
STATE_LESS	
STATE_LESSEQ	
STATE_GREAT	
STATE_GREATEQ	
STATE_EQ	
STATE_NOTEQ	
STATE_INT	
STATE_EOF	
STATE_WR	
STATE_WRI	
STATE_WRIT	
STATE_WRITE	

10.9.3 Function Documentation

10.9.3.1 GetDFAColString()

```
char * GetDFAColString (
            int n )
```

GetDFAColString returns the column name associated with a given number. It is used only for debugging the DFA.

10.9.3.2 GetDFAColumn()

```
short GetDFAColumn (
            char c )
```

Translates a given character into a column index of the state-transition table.

10.9.3.3 GetNextToken()

```
int GetNextToken (
            FILE * file,
            int * charsRead )
```

Gets the next token from the file. Skips leading whitespace. Sets charsRead to the number of characters read, not including whitespace skipped.

Parameters

file	A file to read for tokens.
charsRead	A pointer to an int. The value at charsRead will be overwritten with the number of chars read.

Returns

An int representing a token. See tokens.c.

10.9.3.4 GetNextTokenInBuffer()

```
int GetNextTokenInBuffer (
            char * buffer,
            int * bufIndex,
            int * charsRead )
```

Gets the next token from a buffer. Skips leading whitespace. Sets charsRead to the number of characters read, not including whitespace skipped.

Note

This function primarily exists to test the DFA itself against buffers rather than passing in files.

Parameters

buffer	a character buffer
bufIndex	

10.9.3.5 GetStateString()

```
char * GetStateString (
            int n )
```

10.9.3.6 printCell()

```
void printCell (
            int row,
            int col )
```

A debug function for printing a cell in the DFA.

10.9.3.7 printStateAndChar()

```
void printStateAndChar (
            int row,
            int col )
```

A debug function for printing a state (row name) and column (char name)

10.9.4 Variable Documentation

10.9.4.1 DFA

```
short DFA[71][44][3]
```

The DFA drives the scanner logic. Each of the 71 rows in this state transition table corresponds to a state, or a node on a DFA graph.
Each column corresponds to an edge, with columns 0-25 being 'a' through 'z'. There are also operator characters. The ASCII code of a character is not it's column position and characters must be converted to column numbers before they can index this state transition table. At each cell, there are three values. First is the next state to transition to. Second is the token. Third is a signal to the DFA driver whether to continue reading or not. It says, 'this character is a boundary character for this state'.
The DFA was generated in Excel, and that .xlsx file is available in the /docs folder of this project.

10.10 src/dfa.h File Reference

The DFA and related logic declarations.

```
#include <stdio.h>
```

Functions

- short GetDFAColumn (char c)
- int GetNextToken (FILE ∗file, int ∗charsRead)
- int GetNextTokenInBuffer (char ∗buffer, int ∗bufIndex, int ∗charsRead)
- void printCell (int row, int col)
- void printStateAndChar (int row, int col)

10.10.1 Detailed Description

The DFA and related logic declarations.
The DFA is a 3 dimensional array that maps a given state and character input to a result consisting of the next state, token, and whether reading should continue.
The DFA was created in Excel, and the excel file is available in docs/fullDFA.xlsx.

Authors

Tom Terhune, Karl Miller, Anthony Stepich

Date

February 2023

10.10.2 Function Documentation

10.10.2.1 GetDFAColumn()

```
short GetDFAColumn (
            char c )
```

Translates a given character into a column index of the state-transition table.

10.10.2.2 GetNextToken()

```
int GetNextToken (
            FILE * file,
            int * charsRead )
```

Gets the next token from the file. Skips leading whitespace. Sets charsRead to the number of characters read, not including whitespace skipped.

Parameters

file	A file to read for tokens.
charsRead	A pointer to an int. The value at charsRead will be overwritten with the number of chars read.

Returns

An int representing a token. See tokens.c.

10.10.2.3 GetNextTokenInBuffer()

```
int GetNextTokenInBuffer (
            char * buffer,
            int * bufIndex,
            int * charsRead )
```

Gets the next token from a buffer. Skips leading whitespace. Sets charsRead to the number of characters read, not including whitespace skipped.

Note

This function primarily exists to test the DFA itself against buffers rather than passing in files.

Parameters

buffer	a character buffer
bufIndex	

10.10.2.4 printCell()

```
void printCell (
            int row,
            int col )
```

A debug function for printing a cell in the DFA.

10.10.2.5 printStateAndChar()

```
void printStateAndChar (
            int row,
            int col )
```

A debug function for printing a state (row name) and column (char name)

10.11 dfa.h

Go to the documentation of this file.

```
1 #ifndef dfa_h
2 #define dfa_h
16 #include <stdio.h>
17
21 short GetDFAColumn(char c);
22
29 int GetNextToken(FILE * file, int * charsRead);
30
31
40 int GetNextTokenInBuffer(char * buffer, int * bufIndex, int * charsRead);
41
42
44 void printCell(int row, int col);
45
```

```
47 void printStateAndChar(int row, int col);
48
49 #endif
```

10.12 src/file_util.c File Reference

Functions to assist with file operations.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include "file_util.h"
#include <windows.h>
```

Functions

- short fileExists (const char *filename)
- void backupFile (const char *filename)
- int filenameHasExtension (const char *filename)
- char * addExtension (const char *filename, const char *extension)
- char * removeExtension (const char *filename)
- char * generateAbsolutePath (const char *filename)
- short checkIfSamePaths (const char *filename1, const char *filename2)
- char * getString ()

10.12.1 Detailed Description

Functions to assist with file operations.

Authors

Karl Miller, Tom Terhune, Anthony Stepich

10.12.2 Function Documentation

10.12.2.1 addExtension()

```
char * addExtension (
            const char * filename,
            const char * extension )
```

addExtension modifies the string given by filename by concatenating the string given by extension.
addExtension returns a pointer to a new, concatenated string. This string is allocated with `malloc`. When you are done with it, the memory should be cleared with `free` to avoid memory leaks.

Parameters

filename	the char array to modify
extension	the char array to append

Authors

thomasterh99, klm127

Date

1/18/2023

Note

Covered by Unit Tests

10.12.2.2 backupFile()

```
void backupFile (
            const char * filename )
```

Renames an existing file, adding the extension '.bak' to the end of it. For example 'outFile.out' will become 'out↩ File.out.bak'.
If the backup file exists already, the new file will have additional '.bak's appended until a name is found that does not collide.

Author

klm127

Date

1/22/2023

Note

Covered by Unit Tests

10.12.2.3 checkIfSamePaths()

```
short checkIfSamePaths (
            const char * filename1,
            const char * filename2 )
```

checkIfSamePaths uses generateAbsolutePath to see if two filenames have the same resulting path.

Precondition

both filenames should be validated to be possible filenames.

Parameters

filename1	the first filename to check.
filename2	the second filename to check.

Returns

1 if they are the same path, 0 otherwise.

Author

karl

Date

2/1/2023

10.12.2.4 fileExists()

```
short fileExists (
            const char * filename )
```

fileExists checks whether a file with name filename exists.

Parameters

filename	: the filename to check.

Returns

short:

- 1 if the file exists
- 0 if it does not.
- -1 if file cant exist

Authors

klm127

Date

1/19/2023

Note

Covered by Unit Tests

10.12.2.5 filenameHasExtension()

```
int filenameHasExtension (
            const char * filename )
```

filenameHasExtension checks whether a filename has an extension. It validates that a string would be a valid path but with one additional condition: it must have a period in the file name portion of the path followed by at least one character.

Parameters

filename	the string to check

Returns

int:

- the index of the `.` character in the string if it exists. otherwise, one of the negative `FILE_EXTENSION_PARSE` enums indicating why the filename is invalid;
  - (-1) means there was no period.
  - (-2) means it ended in a period.
  - (-3) means it is only a period.
  - (-4) means it ends in a slash and is a directory.

Author

klm127

Date

1/19/2023

Note

Covered by Unit Tests

10.12.2.6 generateAbsolutePath()

```
char * generateAbsolutePath (
            const char * filename )
```

generateAbsolutePath uses a fileapi.h call to generate the absolute path for a given filename.

Precondition

filename has already been validated to have an extension

Parameters

filename	the filename to create an absolute path for

Returns

a malloced string for a full path name

Warning

ensure the returned string is freed when you are done to avoid memory leaks

Authors

karl, anthony, thomas

Date

2/1/2023

10.12.2.7 getString()

```
char * getString ( )
```

getString scans a string character by character until recieving a null termination character or a new line

Returns

a pointer to a new character array given by the user with a size of the number of characters + 4 for the possible extension This string is allocated with `malloc`. When you are done with it, the memory should be cleared with `free` to avoid memory leaks.

Author

thomaserh99

Date

1/23/2023

Note

Covered by Unit Tests

10.12.2.8 removeExtension()

```
char * removeExtension (
            const char * filename )
```

removeExtension modifices the string given in parameters by copying the characters of the string up to the index of the last period.

Precondition

filename has been validated to have a correct extension (not leading with a '.', not ending with a '.')

Parameters

filename	the filename char* to remove the extension from.

Returns

a pointer to a new, extensionless string.

Warning

This string is allocated with `malloc`. When you are done with it, the memory should be cleared with `free` to avoid memory leaks.

Authors

thomasterh99, klm127

Date

1/22/2023

Note

Covered by Unit Tests

10.13 src/file_util.h File Reference

Functions to assist with file operations.

```
#include <stdbool.h>
#include <stdio.h>
```

Enumerations

- enum FILE_EXISTS_ENUM { FILE_CANT_EXIST = -1 , FILE_EXISTS = 1 , FILE_DOES_NOT_EXIST = 0 }
- enum FILENAME_EXTENSION_PARSE { FILENAME_HAS_NO_PERIOD = -1 , FILENAME_ENDS_IN_PERIOD = -2 , FILENAME_IS_ONLY_PERIOD = -3 , FILENAME_IS_DIRECTORY = -4 }

Functions

- void backupFile (const char *filename)
- short fileExists (const char *filename)
- int filenameHasExtension (const char *filename)
- char * addExtension (const char *filename, const char *extension)
- char * removeExtension (const char *filename)
- char * generateAbsolutePath (const char *filename)
- short checkIfSamePaths (const char *filename1, const char *filename2)
- char * getString ()

10.13.1 Detailed Description

Functions to assist with file operations.

Authors

Karl Miller, Tom Terhune, Anthony Stepich

10.13.2 Enumeration Type Documentation

10.13.2.1 FILE_EXISTS_ENUM

`enum FILE_EXISTS_ENUM`
Alias for true false, 1, 0

Enumerator

FILE_CANT_EXIST	
FILE_EXISTS	
FILE_DOES_NOT_EXIST	

10.13.2.2 FILENAME_EXTENSION_PARSE

`enum FILENAME_EXTENSION_PARSE`
The enum FILENAME_EXTENSION_PARSE describes possible return values from filenameHasExtension which indicate different ways which a filename may be invalid.

Enumerator

FILENAME_HAS_NO_PERIOD	
FILENAME_ENDS_IN_PERIOD	
FILENAME_IS_ONLY_PERIOD	
FILENAME_IS_DIRECTORY	

10.13.3 Function Documentation

10.13.3.1 addExtension()

```
char * addExtension (
            const char * filename,
            const char * extension )
```
addExtension modifies the string given by filename by concatenating the string given by extension.
addExtension returns a pointer to a new, concatenated string. This string is allocated with `malloc`. When you are done with it, the memory should be cleared with `free` to avoid memory leaks.

Parameters

filename	the char array to modify
extension	the char array to append

Authors

thomasterh99, klm127

Date

1/18/2023

Note

Covered by Unit Tests

10.13.3.2 backupFile()

```
void backupFile (
            const char * filename )
```

Renames an existing file, adding the extension '.bak' to the end of it. For example 'outFile.out' will become 'out↩ File.out.bak'.
If the backup file exists already, the new file will have additional '.bak's appended until a name is found that does not collide.

Author

klm127

Date

1/22/2023

Note

Covered by Unit Tests

10.13.3.3 checkIfSamePaths()

```
short checkIfSamePaths (
            const char * filename1,
            const char * filename2 )
```

checkIfSamePaths uses generateAbsolutePath to see if two filenames have the same resulting path.

Precondition

both filenames should be validated to be possible filenames.

Parameters

filename1	the first filename to check.
filename2	the second filename to check.

Returns

1 if they are the same path, 0 otherwise.

Author

karl

Date

2/1/2023

10.13.3.4 fileExists()

```
short fileExists (
            const char * filename )
```

fileExists checks whether a file with name filename exists.

Parameters

filename	: the filename to check.

Returns

short:

- 1 if the file exists
- 0 if it does not.
- -1 if file cant exist

Authors

klm127

Date

1/19/2023

Note

Covered by Unit Tests

10.13.3.5 filenameHasExtension()

```
int filenameHasExtension (
            const char * filename )
```

filenameHasExtension checks whether a filename has an extension. It validates that a string would be a valid path but with one additional condition: it must have a period in the file name portion of the path followed by at least one character.

Parameters

filename	the string to check

Returns

int:

- the index of the `.` character in the string if it exists. otherwise, one of the negative `FILE_EXTENSION_PARSE` enums indicating why the filename is invalid;
  - (-1) means there was no period.
  - (-2) means it ended in a period.
  - (-3) means it is only a period.
  - (-4) means it ends in a slash and is a directory.

Author

klm127

Date

1/19/2023

Note

Covered by Unit Tests

10.13.3.6 generateAbsolutePath()

```
char * generateAbsolutePath (
            const char * filename )
```

generateAbsolutePath uses a fileapi.h call to generate the absolute path for a given filename.

Precondition

filename has already been validated to have an extension

Parameters

filename	the filename to create an absolute path for

Returns

a malloced string for a full path name

Warning

ensure the returned string is freed when you are done to avoid memory leaks

Authors

karl, anthony, thomas

Date

2/1/2023

10.13.3.7 getString()

```
char * getString ( )
```

getString scans a string character by character until recieving a null termination character or a new line

Returns

a pointer to a new character array given by the user with a size of the number of characters + 4 for the possible extension This string is allocated with `malloc`. When you are done with it, the memory should be cleared with `free` to avoid memory leaks.

Author

thomaserh99

Date

1/23/2023

Note

Covered by Unit Tests

10.13.3.8 removeExtension()

```
char * removeExtension (
            const char * filename )
```

removeExtension modifices the string given in parameters by copying the characters of the string up to the index of the last period.

Precondition

filename has been validated to have a correct extension (not leading with a '.', not ending with a '.')

Parameters

filename	the filename char* to remove the extension from.

Returns

a pointer to a new, extensionless string.

Warning

This string is allocated with `malloc`. When you are done with it, the memory should be cleared with `free` to avoid memory leaks.

Authors

thomasterh99, klm127

Date

1/22/2023

Note

Covered by Unit Tests

10.14 file_util.h

Go to the documentation of this file.

```
1 #ifndef file_util_h
2 #define file_util_h
9 #include <stdbool.h>
10 #include <stdio.h>
11
12 /*
13 ----------------
14 file operations
15 ----------------
16 */
17 #pragma region fileops
18
28 void backupFile(const char *filename);
29
31 enum FILE_EXISTS_ENUM
32 {
```

```
33     FILE_CANT_EXIST = -1,
34     FILE_EXISTS = 1,
35     FILE_DOES_NOT_EXIST = 0
36 };
51 short fileExists(const char *filename);
52
53 #pragma endregion fileops
54
55 /*
56 --------------------
57 filename functions
58 --------------------
59 */
60 #pragma region filenames
61
65 enum FILENAME_EXTENSION_PARSE
66 {
67     FILENAME_HAS_NO_PERIOD = -1,
68     FILENAME_ENDS_IN_PERIOD = -2,
69     FILENAME_IS_ONLY_PERIOD = -3,
70     FILENAME_IS_DIRECTORY = -4
71 };
72
91 int filenameHasExtension(const char *filename);
92
105 char *addExtension(const char *filename, const char *extension);
106
123 char *removeExtension(const char *filename);
124
136 char *generateAbsolutePath(const char *filename);
137
150 short checkIfSamePaths(const char *filename1, const char *filename2);
151
152 #pragma endregion filenames
153
154 /*
155 ---------------------------
156 prompt assistance functions
157 ---------------------------
158 */
159 #pragma region prompts
160
172 char *getString();
173
174 #pragma endregion prompts
175
176 #endif
```

10.15 src/main.c File Reference

Program entry point.

```
#include "tompiler.h"
```

Functions

- int main (int argc, char ∗argv[ ])

10.15.1 Detailed Description

Program entry point.

Authors

Anthony Stepich

Tom Terhune

Karl Miller

10.15.2 Program 1 - fileopen

10.15.2.1 Group 3

10.15.2.1.1 CSC 460 - Language Translation

10.15.3 Function Documentation

10.15.3.1 main()

```
int main (
            int argc,
            char * argv[] )
```

Program entry point.

10.16 src/parse.c File Reference

Parser struct, 'methods' definitions including Parse functions.

```
#include "parse.h"
#include <stdio.h>
#include "tokens.h"
#include <string.h>
#include "scan.h"
#include "console.h"
#include "stdarg.h"
```

Functions

- void Parser_Load (FILE *out, FILE *list)
- void Parser_Init ()
- void Parser_DeInit ()
- void Parser_expandBuffer ()
- void Parser_clearBuffer ()
- void Parser_pushToBuffer (char *word, int l_word)
- void Parser_printBufferStatementToOutAndClear ()
- void ParseError_MatchFailed (int expected_token)
- void ParseError_NextTokenFailed (int actual_token, int n_expected,...)
- void ParseError_FunctionFailed (const char *functionName)
- short ParseError_SkipToStatementEnd (int endToken)
- void Parser_PrintErrorSummary ()
- int Parser_GetParseErrCount ()
- short Parse_SystemGoal ()
- short Parse_Program ()
- short Parse_StatementList ()
- short Parse_Statement ()
- short Parse_IfTail ()
- short Parse_IDList ()
- short Parse_ExpressionList ()
- short Parse_Expression ()
- short Parse_Term ()
- short Parse_Factor ()
- short Parse_AddOP ()
- short Parse_MultOP ()
- short Parse_Condition ()
- short Parse_Addition ()
- short Parse_Multiplication ()
- short Parse_Unary ()
- short Parse_LPrimary ()
- short Parse_RelOP ()

Variables

- T_Parser parser

10.16.1 Detailed Description

Parser struct, 'methods' definitions including Parse functions.
Parse is responsible for validating the syntax of an input file. It reads tokens provided by the scanner and validates that their sequence conforms with the rules of the language.
Parse_SystemGoal is the entry point, which should be called only after input and output files are loaded into the scanner and parser. It calls for function corresponding to each unique LHS of a production rule.
If a lexical error is encountered (invalid character), the character is skipped and an error is printed to the listing file. Parsing will continue with the next available character.
If a syntax error is encountered within a statement, tokens will be skipped until a semicolon or other end-of-statement symbol is found and information about that error will be printed as a trace in the console and the files.

Authors

Karl Miller, Tom Terhune, Anthony Stepich

Date

March 2023

10.16.2 Function Documentation

10.16.2.1 Parse_Addition()

```
short Parse_Addition ( )
```

10.16.2.2 Parse_AddOP()

```
short Parse_AddOP ( )
```

Processes the add op, which can be + or -, because they share the same precedence.
Production 18: <add op> -> + Production 19: <add op> -> -

10.16.2.3 Parse_Condition()

```
short Parse_Condition ( )
```

Begins parsing a condition operation.
Production 22: <condition> -> <addition> {<rel op> <addition>}

10.16.2.4 Parse_Expression()

```
short Parse_Expression ( )
```

Parses an expression, which begins the parse for arithmetic sequences with order-of-operations.
Production 12: <expression> -> {<add op> }

10.16.2.5 Parse_ExpressionList()

```
short Parse_ExpressionList ( )
```

Parses an expression list, which is 1 or more expressions. It's used with the WRITE production of Statement.
Production 11: <expr list> -> <expression> {, <expr list>}

10.16.2.6 Parse_Factor()

```
short Parse_Factor ( )
```

Processes a factor into a parenthesized expression, negative factor, id, or intliteral.
Production 14: <factor> -> (<expression>) Production 15: <factor> -> - <factor> Production 16: <factor> -> ID Production 17: <factor> -> INTLITERAL

10.16.2.7 Parse_IDList()

```
short Parse_IDList ( )
```

Parses an ID list, which is 1 or more IDs. It's used with the READ production of Statement.
Production 10: <id list> -> ID {,<id list> }

10.16.2.8 Parse_IfTail()

```
short Parse_IfTail ( )
```

Parses the end of an IF statement, which may be an ELSE or an ENDIF.
Production 7: <IFTail> -> ELSE <StatementList> ENDIF Production 8: <IFTail> -> ENDIF

10.16.2.9 Parse_LPrimary()

```
short Parse_LPrimary ( )
```

10.16.2.10 Parse_Multiplication()

```
short Parse_Multiplication ( )
```

10.16.2.11 Parse_MultOP()

```
short Parse_MultOP ( )
```

Processes the add op, which can be $*$ or /, because they share the same precedence.
Production 20: <mult op> -> $*$ Production 21: <mult op> -> /

10.16.2.12 Parse_Program()

```
short Parse_Program ( )
```

Called by SystemGoal. Parses the program, then matches a SCANEOF token.
Production 1: <program> -> BEGIN <statement list> END

10.16.2.13 Parse_RelOP()

```
short Parse_RelOP ( )
```

10.16.2.14 Parse_Statement()

```
short Parse_Statement ( )
```

Called by Program, parses a list of statements
Production 3: <statement> -> ID := <expression>; Production 4: <statement> -> READ (<id list>); Production 5: <statement> -> WRITE (<expr list>); Production 6: <statement> -> IF (<condition>)THEN <Statement↩ List> <IFTail> Production 9: <statement> -> WHILE (<condition>) {<StatementList>} ENDWHILE

10.16.2.15 Parse_StatementList()

```
short Parse_StatementList ( )
```

Called by Program. Parses a statement, then possibly processes an additional statement list.
Production 2: <statement list> -> <statement> {<statement list>}

10.16.2.16 Parse_SystemGoal()

```
short Parse_SystemGoal ( )
```

Called by main. Begins the parsing process.
Production 40. <system goal> -> <program> SCANEOF

10.16.2.17 Parse_Term()

```
short Parse_Term ( )
```

Continues the inner expression parse by looking for multiplication symbols.
Production 13: -> <factor> {<mult op> <factor>}

10.16.2.18 Parse_Unary()

```
short Parse_Unary ( )
```

10.16.2.19 ParseError_FunctionFailed()

```
void ParseError_FunctionFailed (
            const char * functionName )
```

Prints a parse error for a function failing to the output file and the console. The printing will be indented, with the deepest function left-aligned. This allows tracing of the parse functions that failed.
Increments parser.trace by 1 to allow visual indentation.

10.16.2.20 ParseError_MatchFailed()

```
void ParseError_MatchFailed (
            int expected_token )
```

Prints a parse error for when a match failed.

Parameters

expected_token	The token that failed to match.

10.16.2.21 ParseError_NextTokenFailed()

```
void ParseError_NextTokenFailed (
            int actual_token,
            int n_expected,
             ... )
```

Prints a parse error when a next-token lookahead failed. Prints it indented as much as the current trace

Parameters

actual_token	The actual token that was found.
n_expected	The number of possible expected tokens.
...	The valid expected tokens.

10.16.2.22 ParseError_SkipToStatementEnd()

```
short ParseError_SkipToStatementEnd (
            int endtoken )
```

Attempt ParseError recovery.

Called when a statement has a syntax error. Skips over tokens until it reaches a statement-end token, such as ENDIF, ENDWHILE, or SEMICOLON, depending on the parse situation.
For example, if parsing fails inside a WHILE statement, everything until the next ENDWHILE will be skipped to attempt to recover from the error.
Encountering END or SCANEOF will also terminate the skipping feature.
Skipping lines is noted in the listing file and the number of tokens and lines skipped is printed to the out file and console.
This allows some limited recovery from parse errors.

Parameters

endtoken	The endtoken to skip to.

Returns

0 if it was able to skip to the target endtoken, 1 if it encountered END or SCANEOF before then.

10.16.2.23 Parser_clearBuffer()

```
void Parser_clearBuffer ( )
```

If parser.buffer has been allocated, it is freed. parser.buffer is given memory on the heap equal to the const PARSER_BUFFER_INITIAL_CAPACITY.
Also resets parser.l_buffer to 0 and parser.capacity.

10.16.2.24 Parser_DeInit()

```
void Parser_DeInit ( )
```

Deinitializes the parser, setting file references to NULL (but not closing the files.)

10.16.2.25 Parser_expandBuffer()

```
void Parser_expandBuffer ( )
```

Doubles the size of the buffer, preserving data in the buffer (But possibly moving it to a different location). parser.↩ buffer will point to a buffer twice as large and parser.capacity will be doubled. If necessary, an old buffer may have been freed. Do not use parser's buffer directly as it may be freed.

10.16.2.26 Parser_GetParseErrCount()

```
int Parser_GetParseErrCount ( )
```

Returns the error count of parser.

Returns

The number of errors parser had.

10.16.2.27 Parser_Init()

```
void Parser_Init ( )
```

Performs any initialization needed by the parser.

10.16.2.28 Parser_Load()

```
void Parser_Load (
            FILE * out,
            FILE * list )
```

Parser_Load loads the output and listing files to the parser struct so additional information can be printed to them.
It also loads a pointer to the scanners buffer length, so that it can use the scanner's buffer for its own printing operations.

Parameters

out	The output file
list	The listing file

10.16.2.29 Parser_printBufferStatementToOutAndClear()

```
void Parser_printBufferStatementToOutAndClear ( )
```

Prints the contents of the buffer to parser.out. Prepends "Statement: " to the printed text. Inserts newlines before and after. "Clears" the buffer afterwards, by setting the length to 0.

10.16.2.30 Parser_PrintErrorSummary()

```
void Parser_PrintErrorSummary ( )
```

Prints a summary of the parse errors to the listing file and the console.

10.16.2.31 Parser_pushToBuffer()

```
void Parser_pushToBuffer (
            char * word,
            int l_word )
```

Pushes a char * of length l_word to parser's buffer. Moves parser.l_buffer as necessary. Reallocates the buffer if needed by calling Parser_expandBuffer. Puts a null terminator after the end of valid characters.

Parameters

word	A char array to add to the buffer.
l_word	The length of word, in chars.

10.16.3 Variable Documentation

10.16.3.1 parser

```
T_Parser parser
```

10.17 src/parse.h File Reference

Parser struct, 'methods' declarations including Parse functions.

```
#include <stdio.h>
```

Data Structures

- struct T_Parser

Macros

- #define PARSER_BUFFER_INITIAL_CAPACITY 50

Functions

- void Parser_Load (FILE *out, FILE *list)

- void Parser_Init ()
- void Parser_DeInit ()
- void Parser_expandBuffer ()
- void Parser_clearBuffer ()
- void Parser_pushToBuffer (char *word, int l_word)
- void Parser_printBufferStatementToOutAndClear ()
- void ParseError_MatchFailed (int expected_token)
- void ParseError_NextTokenFailed (int actual_token, int n_expected,...)
- void ParseError_FunctionFailed (const char *functionName)
- short ParseError_SkipToStatementEnd (int endtoken)
- void Parser_PrintErrorSummary ()
- int Parser_GetParseErrCount ()
- short Parse_Program ()
- short Parse_StatementList ()
- short Parse_Statement ()
- short Parse_IfTail ()
- short Parse_IDList ()
- short Parse_ExpressionList ()
- short Parse_Expression ()
- short Parse_Term ()
- short Parse_Factor ()
- short Parse_AddOP ()
- short Parse_MultOP ()
- short Parse_Condition ()
- short Parse_Addition ()
- short Parse_Multiplication ()
- short Parse_Unary ()
- short Parse_LPrimary ()
- short Parse_RelOP ()
- short Parse_SystemGoal ()

10.17.1 Detailed Description

Parser struct, 'methods' declarations including Parse functions.
Parse is responsible for validating the syntax of an input file. It reads tokens provided by the scanner and validates that their sequence conforms with the rules of the language.
Parse_SystemGoal is the entry point, which should be called only after input and output files are loaded into the scanner and parser. It calls for function corresponding to each unique LHS of a production rule.
If a lexical error is encountered (invalid character), the character is skipped and an error is printed to the listing file. Parsing will continue with the next available character.
If a syntax error is encountered within a statement, tokens will be skipped until a semicolon or other end-of-statement symbol is found and information about that error will be printed as a trace in the console and the files.

Authors

Karl Miller, Tom Terhune, Anthony Stepich

Date

March 2023

10.17.2 Macro Definition Documentation

10.17.2.1 PARSER_BUFFER_INITIAL_CAPACITY

```
#define PARSER_BUFFER_INITIAL_CAPACITY 50
```

10.17.3 Function Documentation

10.17.3.1 Parse_Addition()

```
short Parse_Addition ( )
```

10.17.3.2 Parse_AddOP()

```
short Parse_AddOP ( )
```

Processes the add op, which can be + or -, because they share the same precedence.
Production 18: <add op> -> + Production 19: <add op> -> -

10.17.3.3 Parse_Condition()

```
short Parse_Condition ( )
```

Begins parsing a condition operation.
Production 22: <condition> -> <addition> {<rel op> <addition>}

10.17.3.4 Parse_Expression()

```
short Parse_Expression ( )
```

Parses an expression, which begins the parse for arithmetic sequences with order-of-operations.
Production 12: <expression> -> {<add op> }

10.17.3.5 Parse_ExpressionList()

```
short Parse_ExpressionList ( )
```

Parses an expression list, which is 1 or more expressions. It's used with the WRITE production of Statement.
Production 11: <expr list> -> <expression> {, <expr list>}

10.17.3.6 Parse_Factor()

```
short Parse_Factor ( )
```

Processes a factor into a parenthesized expression, negative factor, id, or intliteral.
Production 14: <factor> -> (<expression>) Production 15: <factor> -> - <factor> Production 16: <factor> -> ID Production 17: <factor> -> INTLITERAL

10.17.3.7 Parse_IDList()

```
short Parse_IDList ( )
```

Parses an ID list, which is 1 or more IDs. It's used with the READ production of Statement.
Production 10: <id list> -> ID {,<id list> }

10.17.3.8 Parse_IfTail()

```
short Parse_IfTail ( )
```

Parses the end of an IF statement, which may be an ELSE or an ENDIF.
Production 7: <IFTail> -> ELSE <StatementList> ENDIF Production 8: <IFTail> -> ENDIF

10.17.3.9 Parse_LPrimary()

```
short Parse_LPrimary ( )
```

10.17.3.10 Parse_Multiplication()

```
short Parse_Multiplication ( )
```

10.17.3.11 Parse_MultOP()

```
short Parse_MultOP ( )
```

Processes the add op, which can be ∗ or /, because they share the same precedence.
Production 20: <mult op> -> ∗ Production 21: <mult op> -> /

10.17.3.12 Parse_Program()

```
short Parse_Program ( )
```

Called by SystemGoal. Parses the program, then matches a SCANEOF token.
Production 1: <program> -> BEGIN <statement list> END

10.17.3.13 Parse_RelOP()

```
short Parse_RelOP ( )
```

10.17.3.14 Parse_Statement()

```
short Parse_Statement ( )
```

Called by Program, parses a list of statements
Production 3: <statement> -> ID := <expression>; Production 4: <statement> -> READ (<id list>); Production 5: <statement> -> WRITE (<expr list>); Production 6: <statement> -> IF (<condition>)THEN <Statement↩ List> <IFTail> Production 9: <statement> -> WHILE (<condition>) {<StatementList>} ENDWHILE

10.17.3.15 Parse_StatementList()

```
short Parse_StatementList ( )
```

Called by Program. Parses a statement, then possibly processes an additional statement list.
Production 2: <statement list> -> <statement> {<statement list>}

10.17.3.16 Parse_SystemGoal()

```
short Parse_SystemGoal ( )
```

Called by main. Begins the parsing process.
Production 40. <system goal> -> <program> SCANEOF

10.17.3.17 Parse_Term()

```
short Parse_Term ( )
```

Continues the inner expression parse by looking for multiplication symbols.
Production 13: -> <factor> {<mult op> <factor>}

10.17.3.18 Parse_Unary()

```
short Parse_Unary ( )
```

10.17.3.19 ParseError_FunctionFailed()

```
void ParseError_FunctionFailed (
            const char * functionName )
```

Prints a parse error for a function failing to the output file and the console. The printing will be indented, with the deepest function left-aligned. This allows tracing of the parse functions that failed.
Increments parser.trace by 1 to allow visual indentation.

10.17.3.20 ParseError_MatchFailed()

```
void ParseError_MatchFailed (
            int expected_token )
```

Prints a parse error for when a match failed.

Parameters

expected_token	The token that failed to match.

10.17.3.21 ParseError_NextTokenFailed()

```
void ParseError_NextTokenFailed (
            int actual_token,
            int n_expected,
             ...  )
```

Prints a parse error when a next-token lookahead failed. Prints it indented as much as the current trace

Parameters

actual_token	The actual token that was found.
n_expected	The number of possible expected tokens.
...	The valid expected tokens.

10.17.3.22 ParseError_SkipToStatementEnd()

```
short ParseError_SkipToStatementEnd (
            int endtoken )
```

Attempt ParseError recovery.
Called when a statement has a syntax error. Skips over tokens until it reaches a statement-end token, such as ENDIF, ENDWHILE, or SEMICOLON, depending on the parse situation.
For example, if parsing fails inside a WHILE statement, everything until the next ENDWHILE will be skipped to attempt to recover from the error.
Encountering END or SCANEOF will also terminate the skipping feature.
Skipping lines is noted in the listing file and the number of tokens and lines skipped is printed to the out file and console.
This allows some limited recovery from parse errors.

Parameters

endtoken	The endtoken to skip to.

Returns

0 if it was able to skip to the target endtoken, 1 if it encountered END or SCANEOF before then.

10.17.3.23 Parser_clearBuffer()

```
void Parser_clearBuffer ( )
```

If parser.buffer has been allocated, it is freed. parser.buffer is given memory on the heap equal to the const PARSER_BUFFER_INITIAL_CAPACITY.
Also resets parser.l_buffer to 0 and parser.capacity.

10.17.3.24 Parser_DeInit()

```
void Parser_DeInit ( )
```

Deinitializes the parser, setting file references to NULL (but not closing the files.)

10.17.3.25 Parser_expandBuffer()

```
void Parser_expandBuffer ( )
```

Doubles the size of the buffer, preserving data in the buffer (But possibly moving it to a different location). parser.↩ buffer will point to a buffer twice as large and parser.capacity will be doubled. If necessary, an old buffer may have been freed. Do not use parser's buffer directly as it may be freed.

10.17.3.26 Parser_GetParseErrCount()

```
int Parser_GetParseErrCount ( )
```

Returns the error count of parser.

Returns

The number of errors parser had.

10.17.3.27 Parser_Init()

```
void Parser_Init ( )
```

Performs any initialization needed by the parser.

10.17.3.28 Parser_Load()

```
void Parser_Load (
            FILE * out,
            FILE * list )
```

Parser_Load loads the output and listing files to the parser struct so additional information can be printed to them. It also loads a pointer to the scanners buffer length, so that it can use the scanner's buffer for its own printing operations.

Parameters

out	The output file
list	The listing file

10.17.3.29 Parser_printBufferStatementToOutAndClear()

```
void Parser_printBufferStatementToOutAndClear ( )
```

Prints the contents of the buffer to parser.out. Prepends "Statement: " to the printed text. Inserts newlines before and after. "Clears" the buffer afterwards, by setting the length to 0.

10.17.3.30 Parser_PrintErrorSummary()

```
void Parser_PrintErrorSummary ( )
```

Prints a summary of the parse errors to the listing file and the console.

10.17.3.31 Parser_pushToBuffer()

```
void Parser_pushToBuffer (
            char * word,
            int l_word )
```

Pushes a char * of length l_word to parser's buffer. Moves parser.l_buffer as necessary. Reallocates the buffer if needed by calling Parser_expandBuffer. Puts a null terminator after the end of valid characters.

Parameters

word	A char array to add to the buffer.

Parameters

l_word	The length of word, in chars.

10.18 parse.h

Go to the documentation of this file.

```
1 #ifndef parser_h
2 #define parser_h
19 #include <stdio.h>
20
21 #define PARSER_BUFFER_INITIAL_CAPACITY 50
22 /*
23 ------------------------------
24 Typedef for the parser struct
25 ------------------------------
26 */
27 #pragma region typedefs
28 typedef struct {
30     FILE * out;
32     FILE * list;
34     char * buffer;
36     int capacity;
38     int l_buffer;
40     int errorCount;
42     int trace;
43 } T_Parser;
44
45 #pragma endregion typedefs
46
47 /*
48 --------------------------------
49 Lifecycle methods for the parser
50 --------------------------------
51 */
52 #pragma region lifecycle
61 void Parser_Load(FILE *out, FILE *list);
62
66 void Parser_Init();
67
71 void Parser_DeInit();
72
73 #pragma endregion lifecycle
74
75 /*
76 -------------------
77 Parser buffer
78 -------------------
79 */
80 #pragma region buffer
81
85 void Parser_expandBuffer();
86
93 void Parser_clearBuffer();
94
101 void Parser_pushToBuffer(char * word, int l_word);
102
106 void Parser_printBufferStatementToOutAndClear();
107
108 #pragma endregion buffer
109
110 /*
111 -------------------
112 Parse errors
113 -------------------
114 */
115 #pragma region parse_errors
116
121 void ParseError_MatchFailed(int expected_token);
122
129 void ParseError_NextTokenFailed(int actual_token, int n_expected, ...);
130
136 void ParseError_FunctionFailed(const char * functionName);
137
155 short ParseError_SkipToStatementEnd(int endtoken);
156
160 void Parser_PrintErrorSummary();
161
166 int Parser_GetParseErrCount();
167
168 #pragma endregion parse_errors
```

```
169 /*
170 ------------------------------
171 Production rule parse functions
172 ------------------------------
173 */
174 #pragma region production_rule_parse_functions
175
181 short Parse_Program();
182
188 short Parse_StatementList();
189
200 short Parse_Statement();
201
208 short Parse_IfTail();
209
215 short Parse_IDList();
216
222 short Parse_ExpressionList();
223
229 short Parse_Expression();
230
236 short Parse_Term();
237
246 short Parse_Factor();
247
254 short Parse_AddOP();
255
262 short Parse_MultOP();
263
269 short Parse_Condition();
270
271 /*
272 Each side of a logical operation may have arithmetic operations, and precedence must be maintained.
273
274 Production 23:  <addition> -> <multiplication> {<add op> <multiplication>}
275 */
276 short Parse_Addition();
277
278 /*
279 Each side of a logical operation may have arithmetic operations, and precedence must be maintained.
280
281 Production 24:  <multiplication> -> <unary> { <mult op> <unary>}
282 */
283 short Parse_Multiplication();
284
285 /*
286 Unary operations may NOT or NEGATE a logical outcome.
287
288 Production 25:  <unary> -> !  <unary>
289 Production 26:  <unary> -> - <unary>
290 Production 27:  <unary> -> <lprimary>
291 */
292 short Parse_Unary();
293
294 /*
295 LPrimary allows nesting of further conditions or final condition values, such as false and true.
296
297 Produciton 28:    <lprimary> -> INTLITERAL
298 Produciton 29:    <lprimary> -> ID
299 Produciton 30:    <lprimary  -> ( <condition>)
300 Produciton 31:    <lprimary> -> FALSEOP
301 Produciton 32:    <lprimary> -> TRUEOP
302 Produciton 33:    <lprimary> -> NULLOP
303 */
304 short Parse_LPrimary();
305
306 /*
307 Relop results in the standard logical operators.
308
309 Produciton 34:  <RelOp> -> <
310 Produciton 35:  <RelOp> -> <=
311 Produciton 36:  <RelOp> -> >
312 Produciton 37:  <RelOp> -> >=
313 Produciton 38:  <RelOP> -> =
314 Produciton 39:  <RelOp> -> <>
315 */
316 short Parse_RelOP();
317
323 short Parse_SystemGoal();
324
325 #pragma endregion production_rule_parse_functions
326
327 #endif
```

10.19 src/scan.c File Reference

Scanner struct and 'methods' definitions.

```
#include "dfa.h"
#include "tokens.h"
#include "scan.h"
#include "parse.h"
#include <string.h>
#include <stdlib.h>
#include "console.h"
```

Enumerations

- enum LHEAD_RESULT { LH_CLEAR , LH_NLINE , LH_EOF , LH_COMMENT }

Functions

- void Scanner_Init ()
- void Scanner_LoadFiles (FILE *input, FILE *output, FILE *listing, FILE *temp)
- void Scanner_DeInit ()
- FILE * Scanner_DB_GetInFile ()
- void Scanner_clearBuffer ()
- void Scanner_expandBuffer ()
- void Scanner_bufputc (char c)
- void Scanner_ReadBackToBuffer (int n_chars)
- void Scanner_CopyBuffer (char *destination)
- void Scanner_PrintBuffer (FILE *destination, short print_to_console)
- int * Scanner_GetLBuffPointer ()
- char * Scanner_GetBuffer ()
- short Scanner_Lookahead ()
- void Scanner_AdvanceLine ()
- int Scanner_SkipWhitespace ()
- void Scanner_SkipAllWhitespaceForNextToken ()
- void Scanner_ScanAndPrint (FILE *input, FILE *output, FILE *listing, FILE *temp)
- int Scanner_NextToken ()
- void Scanner_SkipLexError ()
- short Scanner_Match (int target_token)
- void Scanner_PrintLine ()
- void Scanner_BackprintIdentifier (int nchars)
- void Scanner_PrintBufferToOutputFile ()
- void Scanner_PrintTokenFront (int token)
- void Scanner_PrintErrorListing ()
- void Scanner_PrintErrorSummary ()
- void Scanner_PrintParseErrorMessage ()
- int Scanner_GetLexErrCount ()

Variables

- struct Scanner scanner

10.19.1 Detailed Description

Scanner struct and 'methods' definitions.
Scanner is responsible for tokenizing an input file. It uses the dfa defined in dfa.c to do so. It prints lines an errors to a listing file and token results to an output file.

Authors

Tom Terhune, Karl Miller, Anthony Stepich

Date

February 2023

10.19.2 Enumeration Type Documentation

10.19.2.1 LHEAD_RESULT

```
enum LHEAD_RESULT
```

Enumerator

LH_CLEAR	
LH_NLINE	
LH_EOF	
LH_COMMENT	

10.19.3 Function Documentation

10.19.3.1 Scanner_AdvanceLine()

```
void Scanner_AdvanceLine ( )
```

Advances the file pointer until the start of the next line. Increments the line-number counter in scanner.

10.19.3.2 Scanner_BackprintIdentifier()

```
void Scanner_BackprintIdentifier (
            int nchars )
```

Moves the file pointer of scanner.in back nchars and prints that many chars to the output file and possibly the console. Used for printing the actual text of a token.

Parameters

nchars	The number of characters to backprint.

10.19.3.3 Scanner_bufputc()

```
void Scanner_bufputc (
            char c )
```

Puts a character in the buffer. Increments scanner.l_buffer. Calls expandBuffer() if necessary.

10.19.3.4 Scanner_clearBuffer()

```
void Scanner_clearBuffer ( )
```

If scanner.buffer has been allocated, it is freed. Scanner.buffer is given memory on the heap equal to the const SCANNER_BUFFER_INITIAL_CAPACITY.
Also resets scanner.l_buffer to 0 and scanner.capacity.

10.19.3.5 Scanner_CopyBuffer()

```
void Scanner_CopyBuffer (
            char * destination )
```

Copies the contents of scanners buffer to another char array destination. Appends a null terminator '\0' to the end of that buffer as well.

Parameters

destination	The string to copy to.

10.19.3.6 Scanner_DB_GetInFile()

```
FILE * Scanner_DB_GetInFile ( )
```

10.19.3.7 Scanner_DeInit()

```
void Scanner_DeInit ( )
```

De-initializes scanner values, setting file pointers to NULL (but not closing files.)

10.19.3.8 Scanner_expandBuffer()

```
void Scanner_expandBuffer ( )
```

Doubles the size of the buffer, preserving data in the buffer (But possibly moving it to a different location). Scanner.buffer will point to a buffer twice as large and scanner.capacity will be doubled. If necessary, an old buffer may have been freed. Do not use scanner's buffer directly as it may be freed; copy the string to a new buffer when extraction is necessary.

10.19.3.9 Scanner_GetBuffer()

```
char * Scanner_GetBuffer ( )
```

Gets the scanner's buffer.

Returns

A char pointer to the scanner's buffer.

10.19.3.10 Scanner_GetLBuffPointer()

```
int * Scanner_GetLBuffPointer ( )
```

Gets a pointer to the scanner's buffer length.

Returns

An int pointer to the scanner's buffer length

10.19.3.11 Scanner_GetLexErrCount()

```
int Scanner_GetLexErrCount ( )
```

gets the number of lexical errors and returns an integer

Returns

integer number of lexical errors

10.19.3.12 Scanner_Init()

```
void Scanner_Init ( )
```

Initializes scanner values to zero.

10.19.3.13 Scanner_LoadFiles()

```
void Scanner_LoadFiles (
            FILE * input,
            FILE * output,
            FILE * listing,
            FILE * temp )
```

Loads input, output, listing, and temp files for scanner referencing.

Parameters

input	The input file which will be scanned.
output	The output file.
listing	The listing file.
temp	The temp file.

10.19.3.14 Scanner_Lookahead()

```
short Scanner_Lookahead ( )
```

Looks ahead to determine if there are any more tokens on the line, or if there is a comment at the end of the line. Resets the fileposition after looking ahead.
This is an internal method of scanner used to determine how it should scan. It should not be called by external modules.

Returns

0 = Clear to Scan, 1 = Newline next, 2 = EOF next, 3 = Comment next,

10.19.3.15 Scanner_Match()

```
short Scanner_Match (
            int target_token )
```

Consumes the next token in scanner.in.

Todo Checks for newlines and prints to the listing file the line if one is found.

Advances column position.

Returns

0 if token is matched correctly. 1 if the tokens do not match.

10.19.3.16 Scanner_NextToken()

```
int Scanner_NextToken ( )
```

Scans the next token in scanner.in. Moves the file pointer back where it started.

Returns

The next token in the input file.

10.19.3.17 Scanner_PrintBuffer()

```
void Scanner_PrintBuffer (
            FILE * destination,
            short print_to_console )
```

10.19.3.18 Scanner_PrintBufferToOutputFile()

```
void Scanner_PrintBufferToOutputFile ( )
```

Prints the buffer to the output file, expecting a null-terminated string. Also prints it to the console if console printing is enabled.

10.19.3.19 Scanner_PrintErrorListing()

```
void Scanner_PrintErrorListing ( )
```

Prints an error message to the listing file and possibly the console. Example: \nError. & not recognized.

10.19.3.20 Scanner_PrintErrorSummary()

```
void Scanner_PrintErrorSummary ( )
```

Prints the total error count to the listing file and possibly the console.

10.19.3.21 Scanner_PrintLine()

```
void Scanner_PrintLine ( )
```

Prints a line with a line number to the listing file. Will print newlines but will not print EOFs. Resets the file pointer to its original position after printing the line.

10.19.3.22 Scanner_PrintParseErrorMessage()

```
void Scanner_PrintParseErrorMessage ( )
```

Prints that a parse error occurred on the current line and column.
Prints this to the listing file and the console.
This doesn't print any information from the parser, just the current line and column number from the Scanner. (Thus why it's a scanner method instead of a parser one.)

10.19.3.23 Scanner_PrintTokenFront()

```
void Scanner_PrintTokenFront (
            int token )
```

10.19.3.24 Scanner_ReadBackToBuffer()

```
void Scanner_ReadBackToBuffer (
            int n_chars )
```

Reads back a number of characters from scanner.in into the buffer. The file pointer will be in the same position after execution.

10.19.3.25 Scanner_ScanAndPrint()

```
void Scanner_ScanAndPrint (
            FILE * input,
            FILE * listing,
            FILE * output,
            FILE * temp )
```

Scans a file for tokens and prints detailed information to the listing and output files.

Warning

This function was used for validating scanner functionality, it is not used when the program is parsing.

Deprecated

Parameters

input	An input file pointer, already opened for reading.
listing	An listing file pointer, already opened for writing.
output	An output file pointer, already opened for writing.
temp	An temp file pointer, already opened for writing.

10.19.3.26 Scanner_SkipAllWhitespaceForNextToken()

```
void Scanner_SkipAllWhitespaceForNextToken ( )
```

Advances the file pointer until a nonwhitespace character (not space or tab or newline) and returns the number of characters skipped.
It does NOT advance the column number.
This is used for Scanner_NextToken, so that the next token can be checked without advancing the column count or printing the lines (which will only occur on match.)

10.19.3.27 Scanner_SkipLexError()

```
void Scanner_SkipLexError ( )
```

Skips a lexical error.
Also prints the error information to the listing file.

10.19.3.28 Scanner_SkipWhitespace()

```
int Scanner_SkipWhitespace ( )
```

Advances the file pointer until a nonwhitespace character (not space or tab) and returns the number of characters skipped.
While the DFA can skip whitespace independently, using this method allows tracking the number of characters that were skipped to maintain an accurate column number.

Returns

The number of whitespace characters skipped.

10.19.4 Variable Documentation

10.19.4.1 scanner

```
struct Scanner scanner
```

10.20 src/scan.h File Reference

Scanner struct and 'methods' declarations.
`#include <stdio.h>`

Data Structures

- struct Scanner

Macros

- #define SCANNER_PRINTS_LINES_TO_CONSOLE 0
- #define SCANNER_PRINTS_TOKENS_TO_CONSOLE 1
- #define SCANNER_BUFFER_INITIAL_CAPACITY 100

Functions

- void Scanner_Init ()
- void Scanner_LoadFiles (FILE *input, FILE *output, FILE *listing, FILE *temp)
- void Scanner_DeInit ()
- FILE * Scanner_DB_GetInFile ()
- void Scanner_clearBuffer ()
- void Scanner_expandBuffer ()
- void Scanner_bufputc (char c)
- void Scanner_ReadBackToBuffer (int n_chars)
- void Scanner_CopyBuffer (char *destination)
- int * Scanner_GetLBuffPointer ()
- char * Scanner_GetBuffer ()
- void Scanner_ScanAndPrint (FILE *input, FILE *listing, FILE *output, FILE *temp)
- void Scanner_AdvanceLine ()
- void Scanner_SkipAllWhitespaceForNextToken ()
- int Scanner_NextToken ()
- short Scanner_Match (int target_token)
- void Scanner_SkipLexError ()
- void Scanner_PrintLine ()
- void Scanner_BackprintIdentifier (int nchars)
- void Scanner_PrintBufferToOutputFile ()
- void Scanner_PrintTokenFront ()
- void Scanner_PrintErrorListing ()
- void Scanner_PrintErrorSummary ()
- void Scanner_PrintParseErrorMessage ()
- int Scanner_GetLexErrCount ()

10.20.1 Detailed Description

Scanner struct and 'methods' declarations.
Scanner is responsible for tokenizing an input file. It uses the dfa defined in dfa.c to do so. It prints lines an errors to a listing file and token results to an output file.

Authors

Tom Terhune, Karl Miller, Anthony Stepich

Date

February 2023

10.20.2 Macro Definition Documentation

10.20.2.1 SCANNER_BUFFER_INITIAL_CAPACITY

```
#define SCANNER_BUFFER_INITIAL_CAPACITY 100
```

10.20.2.2 SCANNER_PRINTS_LINES_TO_CONSOLE

```
#define SCANNER_PRINTS_LINES_TO_CONSOLE 0
```

10.20.2.3 SCANNER_PRINTS_TOKENS_TO_CONSOLE

```
#define SCANNER_PRINTS_TOKENS_TO_CONSOLE 1
```

10.20.3 Function Documentation

10.20.3.1 Scanner_AdvanceLine()

```
void Scanner_AdvanceLine ( )
```

Advances the file pointer until the start of the next line. Increments the line-number counter in scanner.

10.20.3.2 Scanner_BackprintIdentifier()

```
void Scanner_BackprintIdentifier (
            int nchars )
```

Moves the file pointer of scanner.in back nchars and prints that many chars to the output file and possibly the console. Used for printing the actual text of a token.

Parameters

nchars	The number of characters to backprint.

10.20.3.3 Scanner_bufputc()

```
void Scanner_bufputc (
            char c )
```

Puts a character in the buffer. Increments scanner.l_buffer. Calls expandBuffer() if necessary.

10.20.3.4 Scanner_clearBuffer()

```
void Scanner_clearBuffer ( )
```

If scanner.buffer has been allocated, it is freed. Scanner.buffer is given memory on the heap equal to the const SCANNER_BUFFER_INITIAL_CAPACITY.
Also resets scanner.l_buffer to 0 and scanner.capacity.

10.20.3.5 Scanner_CopyBuffer()

```
void Scanner_CopyBuffer (
            char * destination )
```

Copies the contents of scanners buffer to another char array destination. Appends a null terminator '\0' to the end of that buffer as well.

Parameters

destination	The string to copy to.

10.20.3.6 Scanner_DB_GetInFile()

```
FILE * Scanner_DB_GetInFile ( )
```

10.20.3.7 Scanner_DeInit()

```
void Scanner_DeInit ( )
```

De-initializes scanner values, setting file pointers to NULL (but not closing files.)

10.20.3.8 Scanner_expandBuffer()

```
void Scanner_expandBuffer ( )
```

Doubles the size of the buffer, preserving data in the buffer (But possibly moving it to a different location). Scanner.buffer will point to a buffer twice as large and scanner.capacity will be doubled. If necessary, an old buffer may have been freed. Do not use scanner's buffer directly as it may be freed; copy the string to a new buffer when extraction is necessary.

10.20.3.9 Scanner_GetBuffer()

```
char * Scanner_GetBuffer ( )
```

Gets the scanner's buffer.

Returns

A char pointer to the scanner's buffer.

10.20.3.10 Scanner_GetLBuffPointer()

```
int * Scanner_GetLBuffPointer ( )
```

Gets a pointer to the scanner's buffer length.

Returns

An int pointer to the scanner's buffer length

10.20.3.11 Scanner_GetLexErrCount()

```
int Scanner_GetLexErrCount ( )
```

gets the number of lexical errors and returns an integer

Returns

integer number of lexical errors

10.20.3.12 Scanner_Init()

```
void Scanner_Init ( )
```

Initializes scanner values to zero.

10.20.3.13 Scanner_LoadFiles()

```
void Scanner_LoadFiles (
            FILE * input,
            FILE * output,
            FILE * listing,
            FILE * temp )
```

Loads input, output, listing, and temp files for scanner referencing.

Parameters

input	The input file which will be scanned.
output	The output file.
listing	The listing file.
temp	The temp file.

10.20.3.14 Scanner_Match()

```
short Scanner_Match (
            int target_token )
```

Consumes the next token in scanner.in.

Todo Checks for newlines and prints to the listing file the line if one is found.

Advances column position.

Returns

0 if token is matched correctly. 1 if the tokens do not match.

10.20.3.15 Scanner_NextToken()

```
int Scanner_NextToken ( )
```

Scans the next token in scanner.in. Moves the file pointer back where it started.

Returns

The next token in the input file.

10.20.3.16 Scanner_PrintBufferToOutputFile()

```
void Scanner_PrintBufferToOutputFile ( )
```

Prints the buffer to the output file, expecting a null-terminated string. Also prints it to the console if console printing is enabled.

10.20.3.17 Scanner_PrintErrorListing()

```
void Scanner_PrintErrorListing ( )
```

Prints an error message to the listing file and possibly the console. Example: \nError. & not recognized.

10.20.3.18 Scanner_PrintErrorSummary()

```
void Scanner_PrintErrorSummary ( )
```

Prints the total error count to the listing file and possibly the console.

10.20.3.19 Scanner_PrintLine()

```
void Scanner_PrintLine ( )
```

Prints a line with a line number to the listing file. Will print newlines but will not print EOFs. Resets the file pointer to its original position after printing the line.

10.20.3.20 Scanner_PrintParseErrorMessage()

```
void Scanner_PrintParseErrorMessage ( )
```

Prints that a parse error occurred on the current line and column.
Prints this to the listing file and the console.
This doesn't print any information from the parser, just the current line and column number from the Scanner. (Thus why it's a scanner method instead of a parser one.)

10.20.3.21 Scanner_PrintTokenFront()

```
void Scanner_PrintTokenFront ( )
```

Prints the token output to the output file and possibly the console. Example: \ntoken number: 0 token type: BEGIN actual token: After calling, Scanner_BackprintIdentifier should be called to print the actual token.

10.20.3.22 Scanner_ReadBackToBuffer()

```
void Scanner_ReadBackToBuffer (
            int n_chars )
```

Reads back a number of characters from scanner.in into the buffer. The file pointer will be in the same position after execution.

10.20.3.23 Scanner_ScanAndPrint()

```
void Scanner_ScanAndPrint (
            FILE * input,
            FILE * listing,
            FILE * output,
            FILE * temp )
```

Scans a file for tokens and prints detailed information to the listing and output files.

Warning

This function was used for validating scanner functionality, it is not used when the program is parsing.

Deprecated

Parameters

input	An input file pointer, already opened for reading.
listing	An listing file pointer, already opened for writing.
output	An output file pointer, already opened for writing.
temp	An temp file pointer, already opened for writing.

10.20.3.24 Scanner_SkipAllWhitespaceForNextToken()

```
void Scanner_SkipAllWhitespaceForNextToken ( )
```

Advances the file pointer until a nonwhitespace character (not space or tab or newline) and returns the number of characters skipped.
It does NOT advance the column number.
This is used for Scanner_NextToken, so that the next token can be checked without advancing the column count or printing the lines (which will only occur on match.)

10.20.3.25 Scanner_SkipLexError()

```
void Scanner_SkipLexError ( )
```

Skips a lexical error.
Also prints the error information to the listing file.

10.21 scan.h

Go to the documentation of this file.

```
1 #ifndef scan_h
2 #define scan_h
13 #include <stdio.h>
14
15 /*
16 -------------------
17 Flags
18 -------------------
19 */
20
21 #ifndef SCANNER_PRINTS_LINES_TO_CONSOLE
22 #define SCANNER_PRINTS_LINES_TO_CONSOLE 0
23 #endif
24
25 #ifndef SCANNER_PRINTS_TOKENS_TO_CONSOLE
26 #define SCANNER_PRINTS_TOKENS_TO_CONSOLE 1
27 #endif
28
29 #ifndef SCANNER_BUFFER_INITIAL_CAPACITY
30 #define SCANNER_BUFFER_INITIAL_CAPACITY 100
31 #endif
32
33 /*
34 -------------------
35 Scanner lifecycle
36 -------------------
37 */
38 #pragma region lifecycle
42 struct Scanner {
44     int line_no;
46     int col_no;
48     int errors;
49     /* A buffer, primarily for capturing identifiers.  */
50     char * buffer;
51     /* The current capcity of the buffer */
52     int capacity;
53     /* The length of relevant characters in the buffer, also the write index.    */
54     int l_buffer;
55     /* File pointers.  */
56     FILE * in;
57     FILE * out;
58     FILE * temp;
59     FILE * listing;
60 };
61
63 void Scanner_Init();
64
72 void Scanner_LoadFiles(FILE * input, FILE * output, FILE * listing, FILE * temp);
73
75 void Scanner_DeInit();
76
77 FILE* Scanner_DB_GetInFile();
78
79 #pragma endregion lifecycle
80
81 /*
82 -------------------
83 Scanner buffer
84 -------------------
85 */
86 #pragma region buffer
87
94 void Scanner_clearBuffer();
95
99 void Scanner_expandBuffer();
100
104 void Scanner_bufputc(char c);
105
109 void Scanner_ReadBackToBuffer(int n_chars);
110
111
116 void Scanner_CopyBuffer(char * destination);
117
123 int * Scanner_GetLBuffPointer();
```

```
124
129 char * Scanner_GetBuffer();
130
131
132 #pragma endregion buffer
133
134 /*
135 -------------------
136 Scanning methods
137 -------------------
138 */
139 #pragma region scanning
151 void Scanner_ScanAndPrint(FILE *input, FILE *listing, FILE *output, FILE *temp);
152
164 short Scanner_Lookahead();
165
169 void Scanner_AdvanceLine();
170
180 int Scanner_SkipWhitespace();
181
189 void Scanner_SkipAllWhitespaceForNextToken();
190
197 int Scanner_NextToken();
198
208 short Scanner_Match(int target_token);
209
215 void Scanner_SkipLexError();
216
217 #pragma endregion scanning
218
219 /*
220 -------------------
221 Printing methods
222 -------------------
223 */
224 #pragma region printing
225
229 void Scanner_PrintLine();
230
235 void Scanner_BackprintIdentifier(int nchars);
236
240 void Scanner_PrintBufferToOutputFile();
241
245 void Scanner_PrintTokenFront();
246
250 void Scanner_PrintErrorListing();
251
255 void Scanner_PrintErrorSummary();
256
264 void Scanner_PrintParseErrorMessage();
265
272 int Scanner_GetLexErrCount();
273 #pragma endregion printing
274
275
276 #endif
```

10.22 src/tokens.c File Reference

Token map and related functions.

```
#include "tokens.h"
#include <string.h>
#include <stdio.h>
```

Functions

- const char * Token_GetName (int id)

Variables

- const char * tokensMap [ ]

10.22.1 Detailed Description

Token map and related functions.

The tokensMap maps a given token to a constant string, which is used by Token_GetName() to get the name of a token. The index of a token string in the tokensMap is the same as it's enumerated value. E.G, BEGIN is value 0 and "BEGIN" is at position 0 in the tokensMap array.

Authors

Tom Terhune, Karl Miller, Anthony Stepich

Date

February 2023

10.22.2 Function Documentation

10.22.2.1 Token_GetName()

```
const char * Token_GetName (
            int id )
```

Token_GetName gets a character string representing a token.

Parameters

id	The token ENUM to retrieve.

Returns

const char* A string from a lookup table, e.g, "BEGIN". If the param is not a valid token, then it returns "NULL".

Author

klm127

Date

2/7/2023

Note

Covered By Unit Tests

10.22.3 Variable Documentation

10.22.3.1 tokensMap

```
const char* tokensMap[]
```

TokensMap maps each token to the corresponding string.

Warning

If you change the order in the enum, you must also change the order in this map!

10.23 src/tokens.h File Reference

Token functions declarations.

```
#include <stdlib.h>
```

Data Structures

- struct TokenCatch

Enumerations

- enum TOKEN {
  BEGIN =0 , END , READ , WRITE ,
  IF , THEN , ELSE , ENDIF ,
  WHILE , ENDWHILE , ID , INTLITERAL ,
  FALSEOP , TRUEOP , NULLOP , LPAREN ,
  RPAREN , SEMICOLON , COMMA , ASSIGNOP ,
  PLUSOP , MINUSOP , MULTOP , DIVOP ,
  NOTOP , LESSOP , LESSEQUALOP , GREATEROP ,
  GREATEREQUALOP , EQUALOP , NOTEQUALOP , SCANEOF ,
  ERROR }

Functions

- const char * Token_GetName (int id)
- struct TokenCatch * Token_Catch (short tokenType, char *raw_text_found, int line_found_at, int col_found↩ _at)
- char * Token_GetOpRaw (short tokenType)
- struct TokenCatch * Token_CatchOp (short tokenType, int line_found_at, int col_found_at)
- struct TokenCatch * Token_CatchError (char badChar, int line_found_at, int col_found_at)
- void Token_Destroy (struct TokenCatch *token)

10.23.1 Detailed Description

Token functions declarations.
The tokensMap maps a given token to a constant string, which is used by Token_GetName() to get the name of a token. The index of a token string in the tokensMap is the same as it's enumerated value. E.G, BEGIN is value 0 and "BEGIN" is at position 0 in the tokensMap array.
This file also contains declarations for TokenCatch methods, which are no longer used. In an earlier version of the program, a TokenCatch wrapped a given token with related data and was memory-allocated. The current version does not use TokenCatch, but it is retained here in case we need it for future parsing features.

Authors

Tom Terhune, Karl Miller, Anthony Stepich

Date

February 2023

10.23.2 Enumeration Type Documentation

10.23.2.1 TOKEN

`enum TOKEN`

Enumerator

BEGIN	
END	
READ	
WRITE	
IF	

Enumerator

THEN	
ELSE	
ENDIF	
WHILE	
ENDWHILE	
ID	
INTLITERAL	
FALSEOP	
TRUEOP	
NULLOP	
LPAREN	
RPAREN	
SEMICOLON	
COMMA	
ASSIGNOP	
PLUSOP	
MINUSOP	
MULTOP	
DIVOP	
NOTOP	
LESSOP	
LESSEQUALOP	
GREATEROP	
GREATEREQUALOP	
EQUALOP	
NOTEQUALOP	
SCANEOF	
ERROR	

10.23.3 Function Documentation

10.23.3.1 Token_Catch()

```
struct TokenCatch * Token_Catch (
            short tokenType,
            char * raw_text_found,
            int line_found_at,
            int col_found_at )
```

Token_Catch is called when an actual token has been found. It produces a TokenCatch struct which wraps the token type with other associated data, such as the raw text that was found and the line it was found at.

Parameters

tokenType	A type in enum TOKEN
raw_text_found	A char pointer to the raw text that caused this token to be identified as such.
line_found_at	The line in the file the token was found.
col_found_at	The column at which the token was found.

Returns

A new TokenCatch encapsulating the parameter data.

10.23.3.2 Token_CatchError()

```
struct TokenCatch * Token_CatchError (
            char badChar,
            int line_found_at,
            int col_found_at )
```

Token_CatchError is called when an error is found. Whatever character is passed in will become the 'raw' member of a TokenCatch.

Parameters

tokenType	A type in enum TOKEN
raw_text_found	A char pointer to the raw text that caused this token to be identified as such.
line_found_at	The line in the file the token was found.
col_found_at	The column at which the token was found.

Returns

A pointer to a malloced TokenCatch encapsulating the parameter data.

10.23.3.3 Token_CatchOp()

```
struct TokenCatch * Token_CatchOp (
            short tokenType,
            int line_found_at,
            int col_found_at )
```

Token_Catch_Op is called when an op is found. It still produces a TokenCatch but it infers the text that was found based on the token type rather than needing the raw text, since there is not variation in how the operators can be written.

Parameters

tokenType	A type in enum TOKEN
raw_text_found	A char pointer to the raw text that caused this token to be identified as such.
line_found_at	The line in the file the token was found.
col_found_at	The column at which the token was found.

Returns

A new TokenCatch encapsulating the parameter data.

10.23.3.4 Token_Destroy()

```
void Token_Destroy (
            struct TokenCatch * token )
```

Token Destroy deallocates a token by first freeing the internal 'raw' string, then deallocating the token itself.

Parameters

token	A token to deallocate.

10.23.3.5 Token_GetName()

```
const char * Token_GetName (
            int id )
```

Token_GetName gets a character string representing a token.

Parameters

id	The token ENUM to retrieve.

Returns

const char* A string from a lookup table, e.g, "BEGIN". If the param is not a valid token, then it returns "NULL".

Author

klm127

Date

2/7/2023

Note

Covered By Unit Tests

10.23.3.6 Token_GetOpRaw()

```
char * Token_GetOpRaw (
            short tokenType )
```

Token_GetOpName gets a malloced string for assignment to raw representing what must have been found for an operator text given an enumerated operarator token. If its not one of the operators, it returns ':', which is the one case when a valid operator character was a syntactic error.

Parameters

tokenType	The operator token enumerated id

Returns

A malloced string containing the operator, e.g. "<=".

10.24 tokens.h

Go to the documentation of this file.

```
1 #ifndef tokens_h
2 #define tokens_h
16 #include <stdlib.h>
17
18 enum TOKEN {
19     BEGIN=0, END, READ, WRITE, IF, THEN, ELSE, ENDIF, WHILE, ENDWHILE, ID, INTLITERAL, FALSEOP, TRUEOP,
      NULLOP, LPAREN, RPAREN, SEMICOLON, COMMA, ASSIGNOP, PLUSOP, MINUSOP, MULTOP,DIVOP, NOTOP, LESSOP,
      LESSEQUALOP, GREATEROP, GREATEREQUALOP, EQUALOP, NOTEQUALOP, SCANEOF, ERROR
```

```
20 };
21
30 const char * Token_GetName(int id);
31
37 #pragma region token_catch
38 struct TokenCatch{
39     /* A type corresponding to the TOKEN enum.  */
40     short token;
41     /* The character that was found.  */
42     char * raw;
43     /* The line number it was found on.  */
44     int line_no;
45     /* The column where it started.  */
46     int col_no;
47
48 };
49
58 struct TokenCatch* Token_Catch(short tokenType, char* raw_text_found, int line_found_at, int
      col_found_at);
59
65 char * Token_GetOpRaw(short tokenType);
66
75 struct TokenCatch* Token_CatchOp(short tokenType, int line_found_at, int col_found_at);
76
85 struct TokenCatch* Token_CatchError(char badChar, int line_found_at, int col_found_at);
86
91 void Token_Destroy(struct TokenCatch* token);
92
93 #pragma endregion token_catch
94
95 #endif
```

10.25 src/tompiler.c File Reference

Tompiler lifecycle functions.

```
#include "tompiler.h"
#include "windows.h"
#include <wchar.h>
#include <stdio.h>
#include <stdlib.h>
#include "console.h"
#include "parse.h"
#include "scan.h"
```

Functions

- void Tompiler_Init ()
- void Tompiler_Execute (int argc, char ∗argv[ ])
- void Tompiler_DeInit ()
- void Enable_PrettyPrint ()
- void Tompiler_Hello ()
- void Tompiler_Goodbye ()
- void Tompiler_PrintResult (short had_err_in_parse_system_goal, FILE ∗listing)

Variables

- HANDLE handle

10.25.1 Detailed Description

Tompiler lifecycle functions.
Definitions for Tompiler functions. These are the lifecycle functions (Init, Execute, DeInit) and some associated pretty-printing functions.

Authors

Tom Terhune, Karl Miller, Anthony Stepich

Date

March 2023

10.25.2 Function Documentation

10.25.2.1 Enable_PrettyPrint()

```
void Enable_PrettyPrint ( )
```

Enables pretty printing. (Virtual Terminal Sequences)

10.25.2.2 Tompiler_DeInit()

```
void Tompiler_DeInit ( )
```

Tompiler_DeInit deinitializes Tompiler modules.

10.25.2.3 Tompiler_Execute()

```
void Tompiler_Execute (
            int argc,
            char * argv[] )
```

Tompiler_Execute with command line args

10.25.2.4 Tompiler_Goodbye()

```
void Tompiler_Goodbye ( )
```

Prints the goodbye message.

10.25.2.5 Tompiler_Hello()

```
void Tompiler_Hello ( )
```

Prints the hello message.

10.25.2.6 Tompiler_Init()

```
void Tompiler_Init ( )
```

Tompiler_Init the program by initializing the modules needed by Tompiler modules in the correct order.

10.25.2.7 Tompiler_PrintResult()

```
void Tompiler_PrintResult (
            short err,
            FILE * listing )
```

Prints the compilation result. Prints red if compilation failed, yellow if it succeeded with errors, and red if it failed.

Parameters

err	The result of Parse_SystemGoal; will be 1 if compilation failed, 0 otherwise.
listing	The listing file to print to.

10.25.3 Variable Documentation

10.25.3.1 handle

```
HANDLE handle
```

10.26 src/tompiler.h File Reference

Tompiler lifecycle functions.

```
#include "file_util.h"
#include "compfiles.h"
#include "scan.h"
#include <stdio.h>
#include <string.h>
#include <stdbool.h>
#include <stdlib.h>
```

Functions

- void Tompiler_Init ()
- void Tompiler_Execute (int argc, char ∗argv[])
- void Tompiler_DeInit ()
- void Enable_PrettyPrint ()
- void Tompiler_Goodbye ()
- void Tompiler_Hello ()
- void Tompiler_PrintResult (short err, FILE ∗listing)

10.26.1 Detailed Description

Tompiler lifecycle functions.
Declarations for Tompiler functions. These are the lifecycle functions (Init, Execute, DeInit) and some associated pretty-printing functions.

Authors

Tom Terhune, Karl Miller, Anthony Stepich

Date

March 2023

10.26.2 Function Documentation

10.26.2.1 Enable_PrettyPrint()

```
void Enable_PrettyPrint ( )
```

Enables pretty printing. (Virtual Terminal Sequences)

10.26.2.2 Tompiler_DeInit()

```
void Tompiler_DeInit ( )
```

Tompiler_DeInit deinitializes Tompiler modules.

10.26.2.3 Tompiler_Execute()

```
void Tompiler_Execute (
            int argc,
            char * argv[] )
```

Tompiler_Execute with command line args

10.26.2.4 Tompiler_Goodbye()

```
void Tompiler_Goodbye ( )
```

Prints the goodbye message.

10.26.2.5 Tompiler_Hello()

```
void Tompiler_Hello ( )
```

Prints the hello message.

10.26.2.6 Tompiler_Init()

```
void Tompiler_Init ( )
```

Tompiler_Init the program by initializing the modules needed by Tompiler modules in the correct order.

10.26.2.7 Tompiler_PrintResult()

```
void Tompiler_PrintResult (
            short err,
            FILE * listing )
```

Prints the compilation result. Prints red if compilation failed, yellow if it succeeded with errors, and red if it failed.

Parameters

err	The result of Parse_SystemGoal; will be 1 if compilation failed, 0 otherwise.
listing	The listing file to print to.

10.27 tompiler.h

Go to the documentation of this file.

```
1 #ifndef tompiler_h
2 #define tompiler_h
13 #include "file_util.h"
14 #include "compfiles.h"
15
16 #include "scan.h"
17 #include <stdio.h>
18 #include <string.h>
19 #include <stdbool.h>
20 #include <stdlib.h>
21
25 void Tompiler_Init();
26
30 void Tompiler_Execute(int argc, char* argv[]);
31
35 void Tompiler_DeInit();
36
40 void Enable_PrettyPrint();
41
42
46 void Tompiler_Goodbye();
50 void Tompiler_Hello();
51
57 void Tompiler_PrintResult(short err, FILE * listing);
58
59
60
61
62
63 #endif
```

Index