SPL Compiler

Introduction

This readme will outline the steps taken to create the SPL compiler, it will cover why certain decisions were made. It will also highlight the main features of the compiler. Furthermore, it will outline the folder structure, files contained and extra tests that are included.

Files and Folder Structure

```
| -- /bin                              | Folder containing the compiled binaries.
    | -- compiler.exe                  | The spl compiler.
    | -- lexer.exe                     | This executable will read and print the
tokens within an SPL file.
    | -- parser.exe                    | This executable will parse the tokens
from flex and print the bison debug information.
    | -- tree.exe                      | This executable will parse the tokens
from flex and create and print the parse tree in a nice format.
| -- /scripts
    | -- spl_stringify.py              | A simple python script that takes a
string and outputs the equivilant write statment in SPL with variables for various
characters that aren't defined in the language specification.
| -- /src                              | This directory contains all the source
for the four executables.
    | -- spl.c                         |
    | -- spl.y                         | This is the bison file, it also contains
all the logic for optimisations and error and warning generation.
    | -- spl.l                         | This is the lex file that parses raw
text into lexemes which are fed into bison.
| -- /tests                            | This directory contains all the tests
that were both created and part of the course work files. The details of the tests
are in the tests section of this document.
    | -- /errors
        | -- not_declared.spl          |
        | -- prg_identifier_too_long.spl|
        | -- redeclaration.spl         |
        | -- var_identifier_too_long.spl|
        | -- zero_division.spl         |
    | -- /optimisation                 |
        | -- constant_folding.spl      |
        | -- dead_code.spl             |
        | -- dead_stores.spl           |
    | -- /programs                     |
        | -- calculator.spl            |
        | -- quadratic_solver.spl      |
    | -- /provided                     | These tests were provided as part of the
course work.
        | -- a.spl                     |
        | -- b.spl                     |
        | -- c.spl                     |
```

```
        | -- d.spl                      |
        | -- e.spl                      |
        | -- HelloWorld.spl             |
    | -- /warnings
        | -- limits.spl                 |
    | -- read.spl                       |
    | -- write_expressions.spl          |
| -- 600087ACW2019.pdf                  | This is the assesed course work
description and contains the basic specification for the language.
| -- spl_bnf.txt                        | This is the bnf that was derrived from
the railroad diagrams within 600087ACW2019.pdf.
| -- makefile                           | This is the make file that will build
the executables found in the bin directory.
| -- README.md                          | This is this file.
| -- README.html                        | This is this file exported as a html
webpage.
| -- README.pdf                         | This is this file exported as a PDF.
```

Building The Compiler

Prerequisites

The compiler was developed with the following tools and versions; however, it may be compatible with other versions:

flex:

```
$ flex --version
flex 2.6.4
```

bision:

```
$ bison --version
bison (GNU Bison) 3.0.4
Written by Robert Corbett and Richard Stallman.

Copyright (C) 2015 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

gcc:

```
gcc --version
gcc (GCC) 7.4.0
Copyright (C) 2017 Free Software Foundation, Inc.
```

```
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

make:

```
$ make --version
GNU Make 4.2.1
Built for x86_64-unknown-cygwin
Copyright (C) 1988-2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
```

Build Process

To build the compiler a makefile is provided. All C code is compiled with gcc and the -ansi flag. It has several rules that are listed below:

all:

This will build all the exectuables (compiler.exe, tree.exe, parser.exe and lexer.exe).

flex:

This will generate the lex.yy.c file from spl.l; this provides the lexical analyser which passes tokens to the parser. The products are output to the /src directory.

bision:

This will generate the spl_t.c file from spl.y; this provides a parser that matches the tokens fed by the lexical analyser to the defined rules. The products are output to the /src directory.

lexer:

This will build a program that outputs matched tokens from the input file. The products are output to the /bin directory.

parser:

This will build a program that prints the debug information provided by the bison file as the input file. This debug information shows how tokens are matched and rules are shifted and reduced. The products are output to the /bin directory.

tree:

This will build a program that outputs a parse tree generated from the input file. The products are output to the /bin directory.

compiler:

This will build the actual compiler that outputs C code generated from the compiled input file. The products are output to the /bin directory.

clean:

This will remove all files generated by all rules.

lexer-clean:

This will remove all files generated by the lexer rule.

parser-clean:

This will remove all files generated by the parser rule.

tree-clean:

This will remove all files generated by the tree rule.

compiler-clean:

This will remove all files generated by the clean rule.

Running a Rule

To actually execute a rule the following syntax applies:

```
make [rule]
```

for example

```
make lexer
```

If no rule is provided, the default rule is exectuted. For this make file this is the all rule.

Highlights

This compiler implements the following optimisations:

- Dead store removal.
- Dead code removal.

- Constant folding.
- Division by zero constant checking.

The compiler also provides the following functions/features:

- The compiler also generates more complex code for a read statment which waits for valid input for the expected type and flushes the buffer. More on this can be found in the Read Statement section under the Generated Code section.
- The compiler mangles names to ensure they do not clash. More on this can be found in the Name Mangeling section under the Generated Code section.
- Prevents the use of identifier names that are too long.

It also provides warnings for the following:

- Program names not matching.
- Unexpected semi-colons at the end of the last statement in a statement list.
- Type conversion where an overflow or loss of precision may occur.
- Constants which are greater then the upper and lower values of integers and reals.
- When assignments are redundant.
- The use of uninitialised variables.
- Variables that are never used; either in the entire program or after the last assignment.
- Infinite loops.

Parse Tree

The parse tree is a ternery tree that is made up of a struct called Node. Each node may have upto three children; it also has a type which is defined in the enumeration (NodeIdentifiers). Furthermore, a node may have a symbolic link to a symbol table entry which provides more information about that node. The details of the SymbolTableEntry type can be found in the Symbol Table section.

```
typedef enum _nodeIdentifiers
{
    id_program = 1,
    id_block,
    id_declaration_block,
    id_declaration,
    id_identifier_list,
    id_statement_list,
    id_statement,
    id_assignment_statement,
    id_value,
    id_expression,
    id_term,
    id_write_statement,
    id_output_list,
    id_constant,
    id_type,
    id_integer,
    id_comparator,
    id_read_statement,
```

```
    id_if_statement,
    id_if_else_statement,
    id_conditional,
    id_comparison,
    id_for_statement,
    id_for_statement_is_by_to,
    id_while_statement,
    id_do_statement
} NodeIdentifiers;

typedef struct _node Node;
struct _node
{
    SymbolTableEntry* pSymbolTableEntry;
    NodeIdentifiers byNodeIdentifier;
    Node* pFirstChild;
    Node* pSecondChild;
    Node* pThirdChild;
    Node* pParent;
};
```

Symbol Table

A double linked list was chosen over an array. This removed the limit on the number of symbols and it also allows the tab table to be broken down in to multiple sections i.e. a seperate sub-table for variables, operators, constants and types. This will allow the compiler to be optimised as the language develops and the need to compile larger programs arises. While the current implementation adds little performance benifit especially considering that the programs been compiled are so small implementing this early would make implementing future improvements easier. The other benifit to this is there is no real limit to the details that can be stored and it is very maintainable to add extra symbols. The below sections outline the differnt symbol types.

```
typedef enum _symbolTypes
{
    symbol_id_unknown = 0,
    symbol_id_program,
    symbol_id_variable,
    symbol_id_constant,
    symbol_id_type,
    symbol_id_operator
} SymbolTypes;

typedef struct _symbolTableEntry SymbolTableEntry;
struct _symbolTableEntry
{
    struct _symbolTableEntry* pNextTableEntry;
    struct _symbolTableEntry* pPrevTableEntry;
    unsigned char bySymbolType;
    union
    {
```

```
        ProgramDetails programDetails;
        VariableDetails variableDetails;
        ConstantDetails constantDetails;
        TypeDetails typeDetails;
        OperatorDetails operatorDetails;
    } symbolDetails;

};
```

Program

This symbol stores some metadata about the program.

```
typedef struct _programDetails
{
    char acIdentifier[MAX_IDENTIFIER_LENGTH];
    unsigned int uiVariableCount;
    unsigned int uiStatementCount;
} ProgramDetails;
```

Variables

This symbol stores information about a variable such as its type, the identifer of the variable and information about its usage.

```
typedef struct _variableDetails
{
    char acIdentifier[MAX_IDENTIFIER_LENGTH];
    int iType;
    VariableUsageDetails* pFirstUsage;
    VariableUsageDetails* pLastUsage;
} VariableDetails;
```

Constants

This symbol stores information about a constant.

```
typedef struct _constantDetails
{
    int iType;
    union
    {
        int i;
        double f;
        char c;
    } value;
} ConstantDetails;
```

Operators

This symbol stores information about an operator.

```
typedef enum _operatorTypes
{
    operator_type_unknown = 0,
    operator_type_equality,
    operator_type_not_equal,
    operator_type_less_than,
    operator_type_more_than,
    operator_type_less_equal,
    operator_type_more_equal,
    operator_type_multipulcation,
    operator_type_division,
    operator_type_add,
    operator_type_subtract,
    operator_type_not,
    operator_type_and,
    operator_type_or
} OperatorTypes;

typedef struct _operatorDetails
{
    OperatorTypes operatorType;
} OperatorDetails;
```

Types

This symbol stores information about an type.

```
typedef struct _typeDetails
{
    int iType;
} TypeDetails;
```

Variable Usage Tree

As the parse tree is being generated a double linked list is being created that tracks the usage of each variable. Each variable is then evaluated and checked to ensure they are not undeclared or redeclared and then redundant assignments are removed.

```
typedef enum _variableUsageType
{
    variable_usage_declaration,
```

```
    variable_usage_assignment,
    variable_usage_used
} VariableUsageType;

typedef struct _variableUsageDetails
{
    VariableUsageType usageType;
    unsigned int uiLine;
    unsigned int uiPos;
    Node* pParentNode;
    struct _variableUsageDetails * pNextUsage;
    struct _variableUsageDetails * pPrevUsage;
} VariableUsageDetails;
```

Tests

The following table describes the tests that are within the tests directory. The Expected Output column gives the output assuming the input from the Input column is given.

Test File	Description
errors/not_declared.spl	A basic spl file that tests that errors are generated when a variable is used without being declared.
errors/prg_identifier_too_long.spl	A basic spl file that tests that errors are generated when the program identifier exceedes 50 characters.
errors/redeclaration.spl	A basic spl file that checks that redeclaration errors are generated when a variable is redeclared.
errors/var_identifier_too_long.spl	A basic spl file that tests that errors are generated when a variable identifier exceedes 50 characters.
errors/zero_division.spl	A basic spl file that tests that errors are generated for constant division by zeros.
optimisation/constant_folding.spl	A simple optimisation test that should optimise all constant expressions. The generated c should obey BIDMAS. The generated code should be compared witht the SPL.
optimisation/dead_code.spl	A simple optimisation test that should remove dead code. The generated c should have no printfs involving DEAD# and but should have ALIVE# 1-4.
optimisation/dead_stores.spl	A simple optimisation test that should remove dead stores. The generated C should have no redundant assignments. This will require a manual inspection of the generated C agains the SPL.
programs/calculator.spl	A simple calculator that has a simple menu system allowing the choice of an operator (+, -, /, *, ^) and takes two numbers either integer or real or a mix.

Test File	Description
programs/quadratic_solver.spl	A simple solver that solves a quadratic equation where the coefficients are input one at a time. The expected quadratic format is ax^2 + bx + c = 0.
provided/a.spl	A basic spl file that prints hello.
provided/b.spl	A test that takes two inputs (integers) comparing the first to the second and printing the A or B where A is printed if the first was the larges and B if the second was the larges. It then takes a real and multiplies it by 2.3 outputing the result. It finally reads a character and writes it back to the console.
provided/c.spl	A test that iterates a for loop from 1 to 13 and prints the number if it isn't 7. A do while loop that prints from 1 to 14 but not 6 or 8. A while loop that prints from 0 to 11
provided/d.spl	A test that performs some floating point arithmatic.
provided/e.spl	A test that reads an integer and prints any value x that satisfies x <= 5 or x >= 12. Prints a constant expression (36-1). Performs two loops that iterate and print -1 to -5.
provided/HelloWorld.spl	A basic spl file that prints Hello World. This test uses weak typing to calculate the ascii code for the space character and store it in a char.
warnings/limits.spl	A basic spl file that tests if the correct warnings and caps are placed on values that are to big to be stored in their respective types (CHARACTER is ignored and allowed to overflow with a warning).
read.spl	A basic spl file that has multiple read statements and write statements to allow the testing of the read function that the compiler implements. This should be tested with errornous data to see if it is correctly parsed.
write_expressions.spl	A basic spl file that writes various expressions that contain different combinations of types to test the final types (the hiarachy is real > char > int). This test requires a comparison of the SPL and generated C as well as running the compiled program.

Using the Compiler

The compiler can be run by using a terminal or command prompt instance and changing to the /bin directory. The compiler takes the spl code through the stdin stream. It then outputs the generated C code to the stdout stream and all information, warnings and errors are written to the stderr stream. An example of running a simple hello world program using command line on Windows; the first line shows the compiler being run, the second line shows the actual program written directly into the command line window and the rest shows the output from stdin and stderr:

```
C:\Programming\LanguagesAndCompilers\SplCompiler\bin>compiler.exe
HelloWorld: CODE WRITE('H', 'e', 'l', 'l', 'o', 'W', 'o', 'r', 'l', 'd'); NEWLINE
ENDP HelloWorld.

---- Evaluating Variable Usage ----
#include <stdio.h>

void _spl_flush_stdin()
{
        char c = -1;
        do
        {
                c = getchar();
        } while (c != '\n' && c != ' ' && c != EOF);
}

void prg_HelloWorld()
{
        printf("HelloWorld");
        printf("\n");
}

int main()
{
        prg_HelloWorld();
        return 0;
}
```

Piping Input and Output

As it is not ideal to write entire programs on a single the contents of files can be piped into the compiler. The following example shows a test file from the test directory being compiled. The First line shows the file been piped into the compiler and the rest shows a mix of the stdout and stderr streams.

```
C:\Programming\LanguagesAndCompilers\SplCompiler\bin>compiler.exe < ../tests/a.spl

---- Evaluating Variable Usage ----
#include <stdio.h>

void _spl_flush_stdin()
{
        char c = -1;
        do
        {
                c = getchar();
        } while (c != '\n' && c != ' ' && c != EOF);
}

void prg_ProgA()
{
```

```
        printf("hello");
        printf("\n");
}

int main()
{
        prg_ProgA();
        return 0;
}
```

It would be useful for the generated code to ouput to a C file, this can be done by piping stdout into a file. The following shows the test file constant_folding.spl being piped into the compiler and stdout being piped into constant_folding.c. The first line shows the compiler being run with the two pipes and the rest is information, warnings and errors generated by the compiler.

```
C:\Programming\LanguagesAndCompilers\SplCompiler\bin>compiler.exe <
../tests/constant_folding.spl > ../constant_folding.c
[WARNING] - Line 12 | Position 11 - Unexpected semi-colon at the end of the last
statement within the code block. This will be ignored.
[WARNING] - Line 42 | Position 5 - Unexpected semi-colon at the end of the last
statement within the code block. This will be ignored.

---- Evaluating Variable Usage ----
[WARNING] - Line 43 | Position 26 - Variable int is used at line: 5 | pos: 34
before it has been assigned to. It will have a default value of 1.
[WARNING] - Line 43 | Position 26 - Previous usage of variable int was an
assignment, therefore the assignment at line: 18 | pos: 36 was redundant;
therefore, this statement has been optimised out.
[WARNING] - Line 43 | Position 26 - Previous usage of variable int was an
assignment, therefore the assignment at line: 19 | pos: 21 was redundant;
therefore, this statement has been optimised out.
[WARNING] - Line 43 | Position 26 - Previous usage of variable int was an
assignment, therefore the assignment at line: 20 | pos: 21 was redundant;
therefore, this statement has been optimised out.
[WARNING] - Line 43 | Position 26 - Previous usage of variable int was an
assignment, therefore the assignment at line: 21 | pos: 21 was redundant;
therefore, this statement has been optimised out.
[WARNING] - Line 43 | Position 26 - The variable int is not used after the
assignment statement (line: 23 | pos: 33).
[WARNING] - Line 43 | Position 26 - Previous usage of variable real was an
assignment, therefore the assignment at line: 25 | pos: 24 was redundant;
therefore, this statement has been optimised out.
[WARNING] - Line 43 | Position 26 - Previous usage of variable real was an
assignment, therefore the assignment at line: 26 | pos: 26 was redundant;
therefore, this statement has been optimised out.
[WARNING] - Line 43 | Position 26 - Previous usage of variable real was an
assignment, therefore the assignment at line: 27 | pos: 26 was redundant;
therefore, this statement has been optimised out.
[WARNING] - Line 43 | Position 26 - Previous usage of variable real was an
assignment, therefore the assignment at line: 28 | pos: 28 was redundant;
therefore, this statement has been optimised out.
```

```
[WARNING] - Line 43 | Position 26 - Previous usage of variable real was an
assignment, therefore the assignment at line: 30 | pos: 24 was redundant;
therefore, this statement has been optimised out.
[WARNING] - Line 43 | Position 26 - Previous usage of variable real was an
assignment, therefore the assignment at line: 31 | pos: 24 was redundant;
therefore, this statement has been optimised out.
[WARNING] - Line 43 | Position 26 - Previous usage of variable real was an
assignment, therefore the assignment at line: 32 | pos: 24 was redundant;
therefore, this statement has been optimised out.
[WARNING] - Line 43 | Position 26 - Previous usage of variable real was an
assignment, therefore the assignment at line: 33 | pos: 24 was redundant;
therefore, this statement has been optimised out.
[WARNING] - Line 43 | Position 26 - Previous usage of variable real was an
assignment, therefore the assignment at line: 35 | pos: 24 was redundant;
therefore, this statement has been optimised out.
[WARNING] - Line 43 | Position 26 - Previous usage of variable real was an
assignment, therefore the assignment at line: 36 | pos: 24 was redundant;
therefore, this statement has been optimised out.
[WARNING] - Line 43 | Position 26 - Previous usage of variable real was an
assignment, therefore the assignment at line: 37 | pos: 24 was redundant;
therefore, this statement has been optimised out.
[WARNING] - Line 43 | Position 26 - The variable real is never used.
[WARNING] - Line 43 | Position 26 - Previous usage of variable char was an
assignment, therefore the assignment at line: 40 | pos: 24 was redundant;
therefore, this statement has been optimised out.
[WARNING] - Line 43 | Position 26 - The variable char is never used.

C:\Programming\LanguagesAndCompilers\SplCompiler\bin>cat ../constant_folding.c
#include <stdio.h>

void _spl_flush_stdin()
{
        char c = -1;
        do
        {
                c = getchar();
        } while (c != '\n' && c != ' ' && c != EOF);
}

void prg_ConstantFolding()
{
        int spl_int = 1;
        double spl_real = 1;
        char spl_char = 1;

        for (spl_int = 6; 1 < 0 ? spl_int >= 10 : spl_int <= 10; spl_int += 1)
        {
                printf("%d", spl_int);
                printf("\n");
        }
        printf("%c", (char)'b');
        printf("\n");
        printf("%c", (char)'b');
        spl_int = 1;
```

```
        spl_int = (spl_int + 4);
        spl_real = 4.500000;
        spl_char = 'b';
}

int main()
{
        prg_ConstantFolding();
        return 0;
}
```

Finally the information, warnings and errors generated by the compiler can be piped into there own file by prefixing the '>' with the value 2. The following shows an example of an input file being piped into stdin and stdout and stdin being piped into there files.

```
C:\Programming\LanguagesAndCompilers\SplCompiler\bin>compiler.exe <
../tests/constant_folding.spl > ../constant_folding.c 2>../constant_folding.out

C:\Programming\LanguagesAndCompilers\SplCompiler\bin>cat ../constant_folding.c
#include <stdio.h>

void _spl_flush_stdin()
{
        char c = -1;
        do
        {
                c = getchar();
        } while (c != '\n' && c != ' ' && c != EOF);
}

void prg_ConstantFolding()
{
        int spl_int = 1;
        double spl_real = 1;
        char spl_char = 1;

        for (spl_int = 6; 1 < 0 ? spl_int >= 10 : spl_int <= 10; spl_int += 1)
        {
                printf("%d", spl_int);
                printf("\n");
        }
        printf("%c", (char)'b');
        printf("\n");
        printf("%c", (char)'b');
        spl_int = 1;
        spl_int = (spl_int + 4);
        spl_real = 4.500000;
        spl_char = 'b';
}

int main()
```

```
{
        prg_ConstantFolding();
        return 0;
}

C:\Programming\LanguagesAndCompilers\SplCompiler\bin>cat ../constant_folding.out
[WARNING] - Line 12 | Position 11 - Unexpected semi-colon at the end of the last
statement within the code block. This will be ignored.
[WARNING] - Line 42 | Position 5 - Unexpected semi-colon at the end of the last
statement within the code block. This will be ignored.

---- Evaluating Variable Usage ----
[WARNING] - Line 43 | Position 26 - Variable int is used at line: 5 | pos: 34
before it has been assigned to. It will have a default value of 1.
[WARNING] - Line 43 | Position 26 - Previous usage of variable int was an
assignment, therefore the assignment at line: 18 | pos: 36 was redundant;
therefore, this statement has been optimised out.
[WARNING] - Line 43 | Position 26 - Previous usage of variable int was an
assignment, therefore the assignment at line: 19 | pos: 21 was redundant;
therefore, this statement has been optimised out.
[WARNING] - Line 43 | Position 26 - Previous usage of variable int was an
assignment, therefore the assignment at line: 20 | pos: 21 was redundant;
therefore, this statement has been optimised out.
[WARNING] - Line 43 | Position 26 - Previous usage of variable int was an
assignment, therefore the assignment at line: 21 | pos: 21 was redundant;
therefore, this statement has been optimised out.
[WARNING] - Line 43 | Position 26 - The variable int is not used after the
assignment statement (line: 23 | pos: 33).
[WARNING] - Line 43 | Position 26 - Previous usage of variable real was an
assignment, therefore the assignment at line: 25 | pos: 24 was redundant;
therefore, this statement has been optimised out.
[WARNING] - Line 43 | Position 26 - Previous usage of variable real was an
assignment, therefore the assignment at line: 26 | pos: 26 was redundant;
therefore, this statement has been optimised out.
[WARNING] - Line 43 | Position 26 - Previous usage of variable real was an
assignment, therefore the assignment at line: 27 | pos: 26 was redundant;
therefore, this statement has been optimised out.
[WARNING] - Line 43 | Position 26 - Previous usage of variable real was an
assignment, therefore the assignment at line: 28 | pos: 28 was redundant;
therefore, this statement has been optimised out.
[WARNING] - Line 43 | Position 26 - Previous usage of variable real was an
assignment, therefore the assignment at line: 30 | pos: 24 was redundant;
therefore, this statement has been optimised out.
[WARNING] - Line 43 | Position 26 - Previous usage of variable real was an
assignment, therefore the assignment at line: 31 | pos: 24 was redundant;
therefore, this statement has been optimised out.
[WARNING] - Line 43 | Position 26 - Previous usage of variable real was an
assignment, therefore the assignment at line: 32 | pos: 24 was redundant;
therefore, this statement has been optimised out.
[WARNING] - Line 43 | Position 26 - Previous usage of variable real was an
assignment, therefore the assignment at line: 33 | pos: 24 was redundant;
therefore, this statement has been optimised out.
[WARNING] - Line 43 | Position 26 - Previous usage of variable real was an
assignment, therefore the assignment at line: 35 | pos: 24 was redundant;
```

```
therefore, this statement has been optimised out.
[WARNING] - Line 43 | Position 26 - Previous usage of variable real was an
assignment, therefore the assignment at line: 36 | pos: 24 was redundant;
therefore, this statement has been optimised out.
[WARNING] - Line 43 | Position 26 - Previous usage of variable real was an
assignment, therefore the assignment at line: 37 | pos: 24 was redundant;
therefore, this statement has been optimised out.
[WARNING] - Line 43 | Position 26 - The variable real is never used.
[WARNING] - Line 43 | Position 26 - Previous usage of variable char was an
assignment, therefore the assignment at line: 40 | pos: 24 was redundant;
therefore, this statement has been optimised out.
[WARNING] - Line 43 | Position 26 - The variable char is never used.
```

Compiler Flags

A range of flags can be passed to the compiler to disable certain features; these features are enabled by default. The flags are shown in the table below.

Flag	Type	Description
--noOptimisations	Optimisation	This flag disables all optimisations. Pairing with other optimisation flags will have no effects.
--noDeadStores	Optimisation	This flag disables dead store optimisation.
--noConstantFolding	Optimisation	This flag disables constant folding optimisation.
--noDeadCode	Optimisation	This flag disables dead code optimisation.
--noStaticErrors	Static Errors	This flag disables all static error checking.
--noDivisionByZeroCheck	Static Errors	This flag disables static division by a zero constant checking.
-v, --version	Misc.	This flag shows the current version of the compiler that is being used and the data and time that it was built. It then exits the compiler.
-h, --help	Misc.	This flag displays the flags that can be passed to the compiler and what they do.

The following is an example how to use the compiler flags and the different code generated when compiling with the --noDeadStores and --noConstantFolding verses no flags:

Dead Stores and Constant Folding Optimisation Disabled

```
C:\Programming\LanguagesAndCompilers\SplCompiler\bin>compiler.exe --noDeadStores -
-noConstantFolding  < ../tests/constant_folding.spl > ../constant_folding.c 2>
../constant_folding.out

C:\Programming\LanguagesAndCompilers\SplCompiler\bin>cat ../constant_folding.c
/*Dead store optimisation is disabled*/
```

```
/*Constant folding is disabled*/
#include <stdio.h>

void _spl_flush_stdin()
{
        char c = -1;
        do
        {
                c = getchar();
        } while (c != '\n' && c != ' ' && c != EOF);
}

void prg_ConstantFolding()
{
        int spl_int = 1;
        double spl_real = 1;
        char spl_char = 1;

        int _spl_integer_by = ((1 + 1) - 1);
        for (spl_int = ((1 + 2) + 3); ((1 + 1) - 1) < 0 ? spl_int >= 10 : spl_int
<= 10; spl_int += _spl_integer_by)
        {
                printf("%d", spl_int);
                printf("\n");
                _spl_integer_by = ((1 + 1) - 1);
        }
        printf("%c", (char)('a' + 1));
        printf("\n");
        printf("%c", (char)(1 + 'a'));
        spl_int = (((2 + 2) - ((4 * 2) / 2)) + 2);
        spl_int = (2 * 2);
        spl_int = (2 + 2);
        spl_int = (2 - 2);
        spl_int = (2 / 2);
        spl_int = (spl_int + (2 * 2));
        spl_real = (0.500000 * 5);
        spl_real = (2.500000 + 2.500000);
        spl_real = (2.500000 - 2.500000);
        spl_real = (1.250000 / 1.250000);
        spl_real = (2.500000 * 2);
        spl_real = (2.500000 - 2);
        spl_real = (2.500000 / 2);
        spl_real = (2.500000 + 2);
        spl_real = (2 * 2.500000);
        spl_real = (2 - 2.500000);
        spl_real = (2 / 0.500000);
        spl_real = (2 + 2.500000);
        spl_char = ('a' + 1);
        spl_char = ('c' - 1);
}

int main()
{
        prg_ConstantFolding();
```

```
        return 0;
}
```

Dead Stores and Constant Folding Optimisation Enabled

```
C:\Programming\LanguagesAndCompilers\SplCompiler\bin>compiler.exe <
../tests/constant_folding.spl > ../constant_folding.c 2> ../constant_folding.out

C:\Programming\LanguagesAndCompilers\SplCompiler\bin>cat ../constant_folding.c
#include <stdio.h>

void _spl_flush_stdin()
{
        char c = -1;
        do
        {
            c = getchar();
        } while (c != '\n' && c != ' ' && c != EOF);
}

void prg_ConstantFolding()
{
        int spl_int = 1;
        double spl_real = 1;
        char spl_char = 1;

        for (spl_int = 6; 1 < 0 ? spl_int >= 10 : spl_int <= 10; spl_int += 1)
        {
                printf("%d", spl_int);
                printf("\n");
        }
        printf("%c", (char)'b');
        printf("\n");
        printf("%c", (char)'b');
        spl_int = 1;
        spl_int = (spl_int + 4);
        spl_real = 4.500000;
        spl_char = 'b';
}

int main()
{
        prg_ConstantFolding();
        return 0;
}
```

Generated Code

Read Statment

The generated code for the read statement generates code that calls the following function:

```
void _spl_read(const char* pFormat, void* pValue)
{
    while (scanf(pFormat, pValue) != 1)
    {
        getchar();
    };
    _spl_flush_stdin();
}
```

The rationale behind this is to ensure that the devloper who write the SPL gets valid input with minimal effort. The first line of the function:

```
while (scanf(pFormat, pValue) != 1)
```

Loops unit the format is matched (one as a constant as the code that is generated will only ever expect one input). The body of the loop flushes the buffer by getting one character at a time (this removes the character that scanf failed on from the input stream):

```
getchar();
```

When the loop has finished the buffer _spl_flush_stdin is called flushing the buffer until the next delimiter is read ('\n' or ' ') or it reaches the end of the stream (EOF). By using this function valid input can always be retrived for example take the following spl code:

```
read:
    DECLARATIONS
        int OF TYPE INTEGER;
        real OF TYPE REAL;
        char, char2 OF TYPE CHARACTER;
        space, period OF TYPE CHARACTER;
    CODE
        32 -> space;
        46 -> period;
        WRITE('P', 'l', 'e', 'a', 's', 'e', space, 'e', 'n', 't', 'e', 'r', space,
'a', 'n', space, 'i', 'n', 't', 'e', 'g', 'e', 'r', period);
        NEWLINE;
        READ(int);
        WRITE('P', 'l', 'e', 'a', 's', 'e', space, 'e', 'n', 't', 'e', 'r', space,
'a', space, 'r', 'e', 'a', 'l', period);
        NEWLINE;
        READ(real);
        WRITE('P', 'l', 'e', 'a', 's', 'e', space, 'e', 'n', 't', 'e', 'r', space,
'a', space, 'c', 'h', 'a', 'r', 'a', 'c', 't', 'e', 'r', period);
```

```
        NEWLINE;
        READ(char);
        WRITE('P', 'l', 'e', 'a', 's', 'e', space, 'e', 'n', 't', 'e', 'r', space,
'a', space, 'c', 'h', 'a', 'r', 'a', 'c', 't', 'e', 'r', period);
        NEWLINE;
        READ(char2);
        NEWLINE;
        WRITE(int);
        NEWLINE;
        WRITE(real);
        NEWLINE;
        WRITE(char);
        NEWLINE;
        WRITE(char2)
ENDP read.
```

The developer clearly wants to read an int followed by a real etc. By generating the code using the _spl_read function the following the input will be parsed to extract valid input and flush the buffer afterwards:

```
$ ./read.exe
Please enter an integer.
jlkfsdkljsdflkjsdflkj kjplfsdkjplsdflkjlkjfsd32lkjsdfjlkfsdjkfsdjkloo
Please enter a real.
lkjsdflkjsdflkjfsd2.34oikjfsdojikfdsoikj
Please enter a character.
c
Please enter a character.
1

32
2.340000
c
1
```

However, by taking advantage of ' ' being a delimiter multiple inputs can be provided on one line. Here is an example:

```
$ ./read.exe
Please enter an integer.
hkjdfshkjdsjkfshkjsjkshdf hjsdfhjfsdjhkfsdjkhsdjhksdfkj dfsoofsdoijsdfosdfosd
aslkjndlkjasdlkj124elkfadnlf asdsadas2.3asdasdsa ac a
Please enter a real.
Please enter a character.
Please enter a character.

124
2.300000
a
a
```

Name Mangeling

When the compiler parses an identifier it is given a prefix of "spl_" (this is not included in the maximum length of 50 characters as the array length is actually 55 characters). If this identifier is used for the program the prefix is changed to "prg_". By performing this name mangeling the SPL developer can use identifier that are actually keywords in C i.e. int or char. By changing the prefix for a program identifier, the SPL developer can use an identifier for both the program and a variable. Any functions or variables that are provided by the compiler are prefixed with "_spl_". This prevents the developer from being able to name a variable something that would clash with a compiler variable or function while allowing it to be identifier as an SPL generated feature.

Known Issues

- Line numbers for some warnings and errors are incorrect. This is due to global variables being used that are incremented during lexical analysis; therefore, when error checking passes are made the file numbers are not correct as they are already at the end of the file.