18CSC304J - Compiler Design
Mini Project
Semester VI

Simple-C Based Compiler

<u>Team Members</u>

Muralidharan M (RA1911030010031)

Keshav Balaji Babu (RA1911030010005)

Abstract

Compiler is a program that reads a program written in one language, called source language, and translates it into an equivalent program in another language, called target language. It reports errors detected during the translation of source code to target code.

Introduction

There are Four phases in a compiler:

1. Lexical Analyzer

It is also called a scanner. It takes the output of the preprocessor (which performs file inclusion and macro expansion) as the input which is in a pure high-level language. It reads the characters from the source program and groups them into lexemes (sequence of characters that "go together"). Each lexeme corresponds to a token. Tokens are defined by regular expressions which are understood by the lexical analyzer. It also removes lexical errors (e.g., erroneous characters), comments, and white space.

2. Syntax Analyzer

It is sometimes called a parser. It constructs the parse tree. It takes all the tokens one by one and uses Context-Free Grammar to construct the parse tree.
The rules of programming can be entirely represented in a few productions. Using these productions we can represent what the program actually is. The input has to be checked whether it is in the desired format or not.
The parse tree is also called the derivation tree. Parse trees are generally constructed to check for ambiguity in the given grammar. There are certain rules associated with the derivation tree.

- Any identifier is an expression
- Any number can be called an expression
- Performing any operations in the given expression will always result in an expression. For example, the sum of two expressions is also an expression.
- The parse tree can be compressed to form a syntax tree

Syntax error can be detected at this level if the input is not in accordance with the grammar.

3. Semantic Analyzer

It verifies the parse tree, whether it's meaningful or not. It furthermore produces a verified parse tree. It also does type checking, Label checking, and Flow control checking.

4. Intermediate Code Generator
It generates intermediate code, which is a form that can be readily executed by a machine. We have many popular intermediate codes. Example – Three address codes etc. Intermediate code is converted to machine language using the last two phases which are platform dependent. Till intermediate code, it is the same for every compiler out there, but after that, it depends on the platform. To build a new compiler we don't need to build it from scratch. We can take the intermediate code from the already existing compiler and build the last two parts.

Code Optimizer – It transforms the code so that it consumes fewer resources and produces more speed. The meaning of the code being transformed is not altered. Optimization can be categorized into two types: machine-dependent and machine-independent.
Target Code Generator – The main purpose of the Target Code generator is to write a code that the machine can understand and also register allocation, instruction selection, etc. The output is dependent on the type of assembler. This is the final stage of compilation. The optimized code is converted into relocatable machine code which then forms the input to the linker and loader.

This project is a mini C compiler that uses LEX and YACC to generate the Lexical and Syntax analysis.

Functionality
Below is a list containing the different tokens that are identified by our Flex program. It also gives a detailed description of how the different tokens are identified and how errors are detected if any.

Keywords:
The keywords identified are: **int, long, short, long long, signed, unsigned, for, break, continue, if, else, return.**

Identifiers:
Identifiers are identified and added to the symbol table. The rule followed is represented by the regular expression **(_|{letter})({letter}|{digit}|_){0,31}**.
The rule only identifies those lexemes as identifiers which either begin with a letter or an underscore and is followed by either a letter, digit or an underscore with a maximum length of 32.

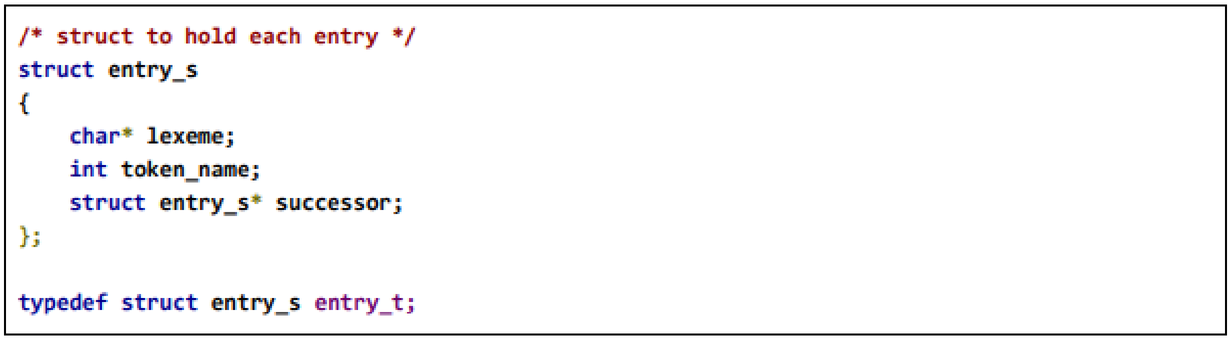
Comments:
Single and multi line comments are identified. Single line comments are identified by //.* regular expression.

Strings:
The lexer can identify strings in any C program. It can also handle double quotes that are escaped using a \ inside a string. Further, error messages are displayed for unterminated strings.

Integer Constants:
The Flex program can identify two types of numeric constants: decimal and hexadecimal. The regular expressions for these are **[+-]?{digit}+[lLuU]**? and **[+-]?0[xX]{hex}+[lLuU]?** respectively.

Preprocessor Directives:
The filenames that come after the **#include** are selectively identified through the exclusive state **<PREPROC>** since the regular expressions for treating the filenames must be kept different from other regexes. Upon encountering a **#include** at the beginning of a line, the lexer switches to the state **<PREPROC>** where it can tokenize filenames of the form “stdio.h” or <stdio.h> . Filenames of any other format are considered as illegal and an error message regarding the same is printed.

SYMBOL TABLE and CONSTANTS TABLE:
We implement a generic hash table with chaining than can be used to declare both a symbol table and a constant table. Every entry in the hash table is a struct of the following form:



```
/* struct to hold each entry */
struct entry_s
{
    char* lexeme;
    int token_name;
    struct entry_s* successor;
};

typedef struct entry_s entry_t;
```

The struct consists of a character pointer to the lexeme that is matched by the lexer, an integer token that is associated with each type of token as defined in **“tokens.h”** and a pointer to the next node in the case of chaining in the hash table. A symbol table or a constant table can be created using the **create_table()** function. The function returns a pointer to a new created hash table which is basically an array of pointers of the type **entry_t***

Every time the lexer matches a pattern, the text that matches the pattern (lexeme) is entered into the associated hash table using an **insert()** function. There are two hash tables maintained: the symbol table and the constants table. Depending on whether the lexeme is a constant or a symbol, an appropriate parameter is passed to the insert function. For example, **insert(symbol_table, yytext, INT)** inserts the keyword **INT** into the symbol table and **insert(constant_table, yytext, HEX_CONSTANT**) inserts a hexadecimal constant into the constants table. The values associated with **INT, HEX_CONSTANT** and other tokens are defined in the tokens.h file. A hash is generated using the matched pattern string as input. We use the Jenkins hash function . The hash table has a fixed size as defined by the user using **HASH_TABLE_SIZE**. The generated hash value is mapped to a value in the range **[0, HASH_TABLE_SIZE)** through the operation hash_value **% HASH_TABLE_SIZE**. This is the index in the hash table for this particular entry. In case the indices clash, a linked list is created and the multiple clashing entries are chained together at that index.

Code Organisation

The entire code for lexical analysis is broken down into 3 files: lexer.l, tokens.h and symboltable.h

File	**Contents**
lexer.l	A lex file containing the lex specification of regular expressions
tokens.h	Contains enumerated constants for keywords, operator, special symbols, constants and identifiers.
symboltable.h	Contains the definition of the symbol table and the constants table and also defines functions for inserting into the hash table and displaying its contents.

Software Requirements:

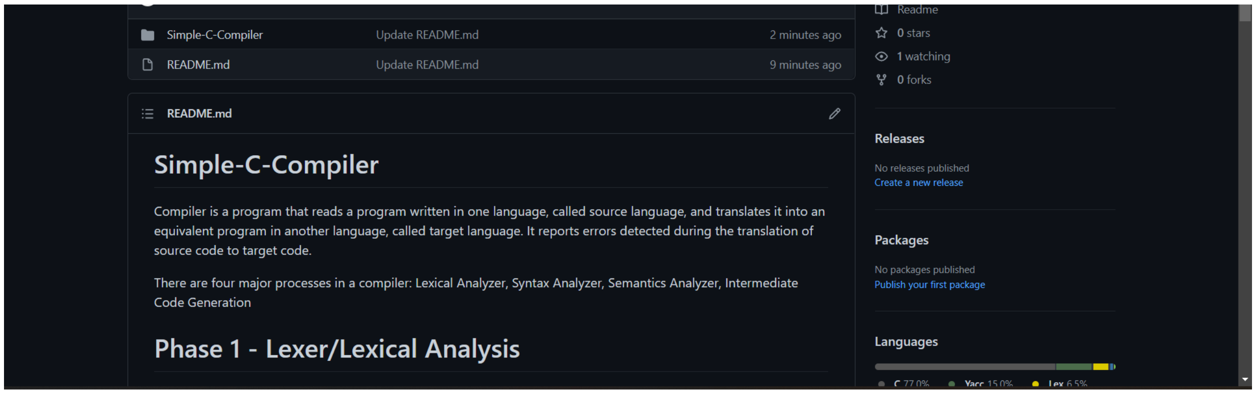
1. Linux/Unix (Recommended)
2. Flex (for lex)
    a. sudo apt-get install flex
3. Bison (for yacc)
    a. sudo apt-get install bison
4. GCC
    a. sudo apt-get install gcc

Hardware Requirements:

1. Intel i3 6th Gen or above
2. RAM: Minimum 4GB
3. Storage: Minimum 2GB

<u>Project Code</u>

View our repository at https://github.com/sherlocked-8601/Simple-C-Compiler



Simple-C-Compiler | Update README.md | 2 minutes ago
README.md | Update README.md | 9 minutes ago

README.md

Simple-C-Compiler

Compiler is a program that reads a program written in one language, called source language, and translates it into an equivalent program in another language, called target language. It reports errors detected during the translation of source code to target code.

There are four major processes in a compiler: Lexical Analyzer, Syntax Analyzer, Semantics Analyzer, Intermediate Code Generation

Phase 1 - Lexer/Lexical Analysis

Readme
0 stars
1 watching
0 forks

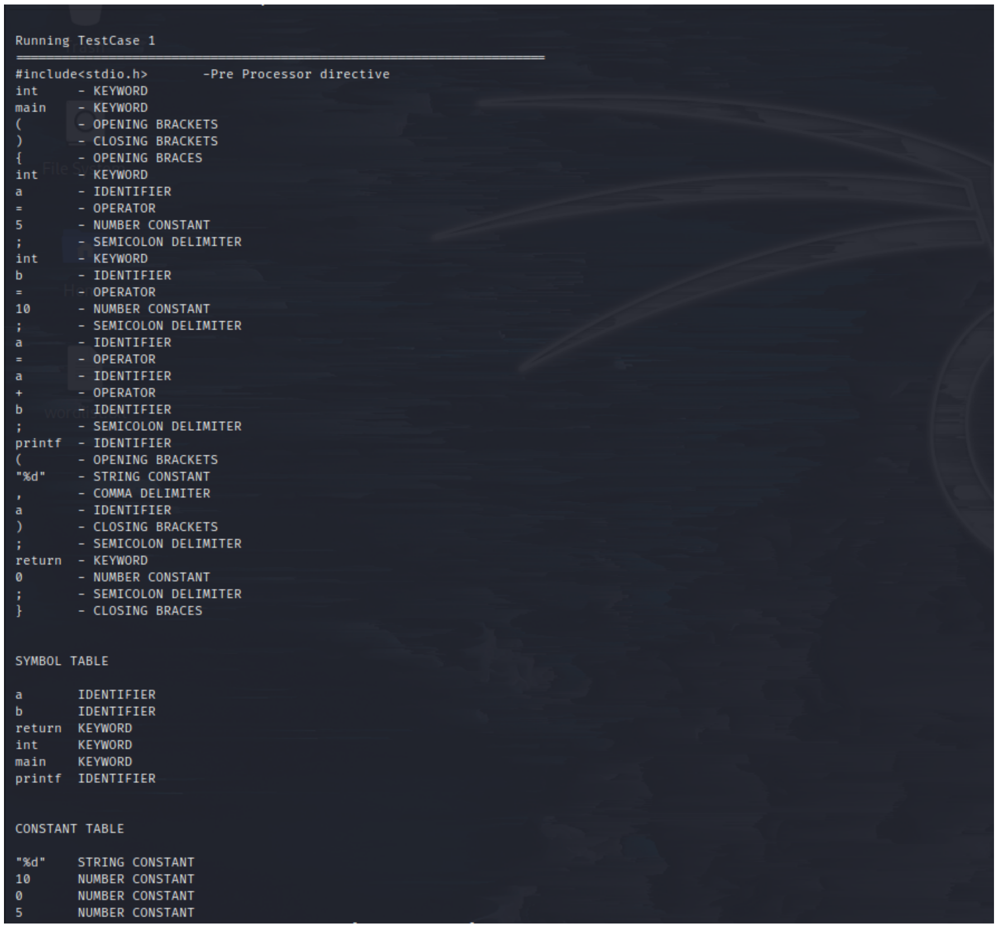
Releases
No releases published
Create a new release

Packages
No packages published
Publish your first package

Languages
C 77.0% Yacc 15.0% Lex 6.5%

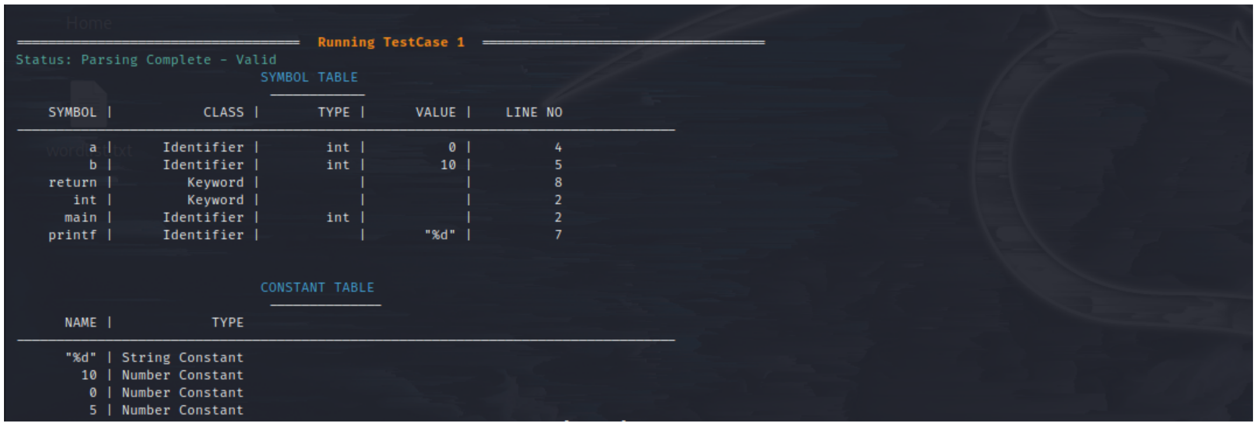
<u>Project Screenshots</u>

1. **Lexical Analyser**



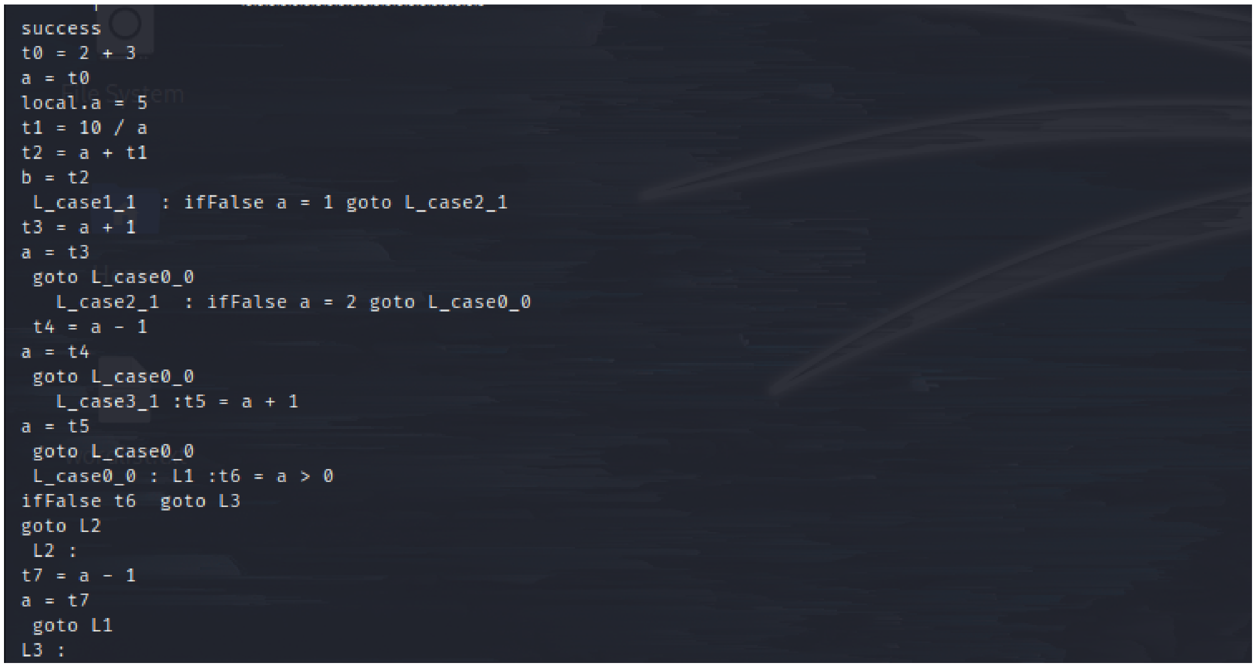
```
Running TestCase 1
==================================================================
#include<stdio.h>       -Pre Processor directive
int      - KEYWORD
main     - KEYWORD
(        - OPENING BRACKETS
)        - CLOSING BRACKETS
{        - OPENING BRACES
int      - KEYWORD
a        - IDENTIFIER
=        - OPERATOR
5        - NUMBER CONSTANT
;        - SEMICOLON DELIMITER
int      - KEYWORD
b        - IDENTIFIER
=        - OPERATOR
10       - NUMBER CONSTANT
;        - SEMICOLON DELIMITER
a        - IDENTIFIER
=        - OPERATOR
a        - IDENTIFIER
+        - OPERATOR
b        - IDENTIFIER
;        - SEMICOLON DELIMITER
printf   - IDENTIFIER
(        - OPENING BRACKETS
"%d"     - STRING CONSTANT
,        - COMMA DELIMITER
a        - IDENTIFIER
)        - CLOSING BRACKETS
;        - SEMICOLON DELIMITER
return   - KEYWORD
0        - NUMBER CONSTANT
;        - SEMICOLON DELIMITER
}        - CLOSING BRACES


SYMBOL TABLE

a       IDENTIFIER
b       IDENTIFIER
return  KEYWORD
int     KEYWORD
main    KEYWORD
printf  IDENTIFIER


CONSTANT TABLE

"%d"    STRING CONSTANT
10      NUMBER CONSTANT
0       NUMBER CONSTANT
5       NUMBER CONSTANT
```

2. Parse Table



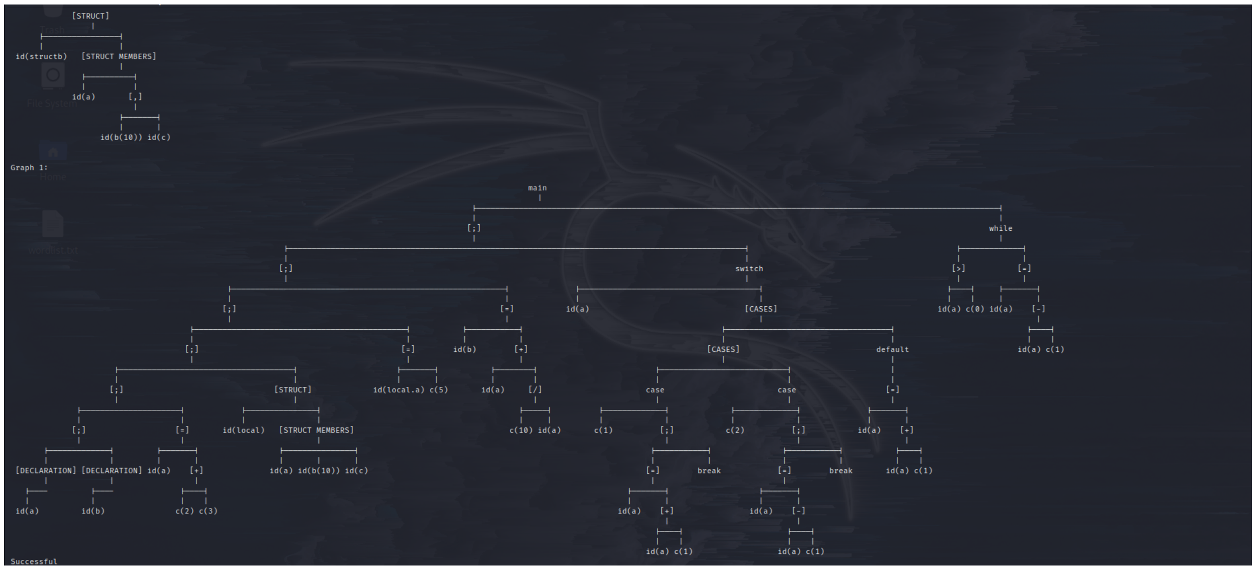
```
====================================  Running TestCase 1  ====================================
Status: Parsing Complete - Valid
                                SYMBOL TABLE
                                _____________

    SYMBOL |            CLASS |        TYPE |      VALUE |     LINE NO
_____________________________________________________________________________________
         a |       Identifier |         int |          0 |           4
         b |       Identifier |         int |         10 |           5
    return |          Keyword |             |            |           8
       int |          Keyword |             |            |           2
      main |       Identifier |         int |            |           2
    printf |       Identifier |             |       "%d" |           7


                                CONSTANT TABLE
                                ______________

      NAME |             TYPE
_____________________________________________________________________________________

      "%d" | String Constant
        10 | Number Constant
         0 | Number Constant
         5 | Number Constant
```

3. Inserting Temporaries



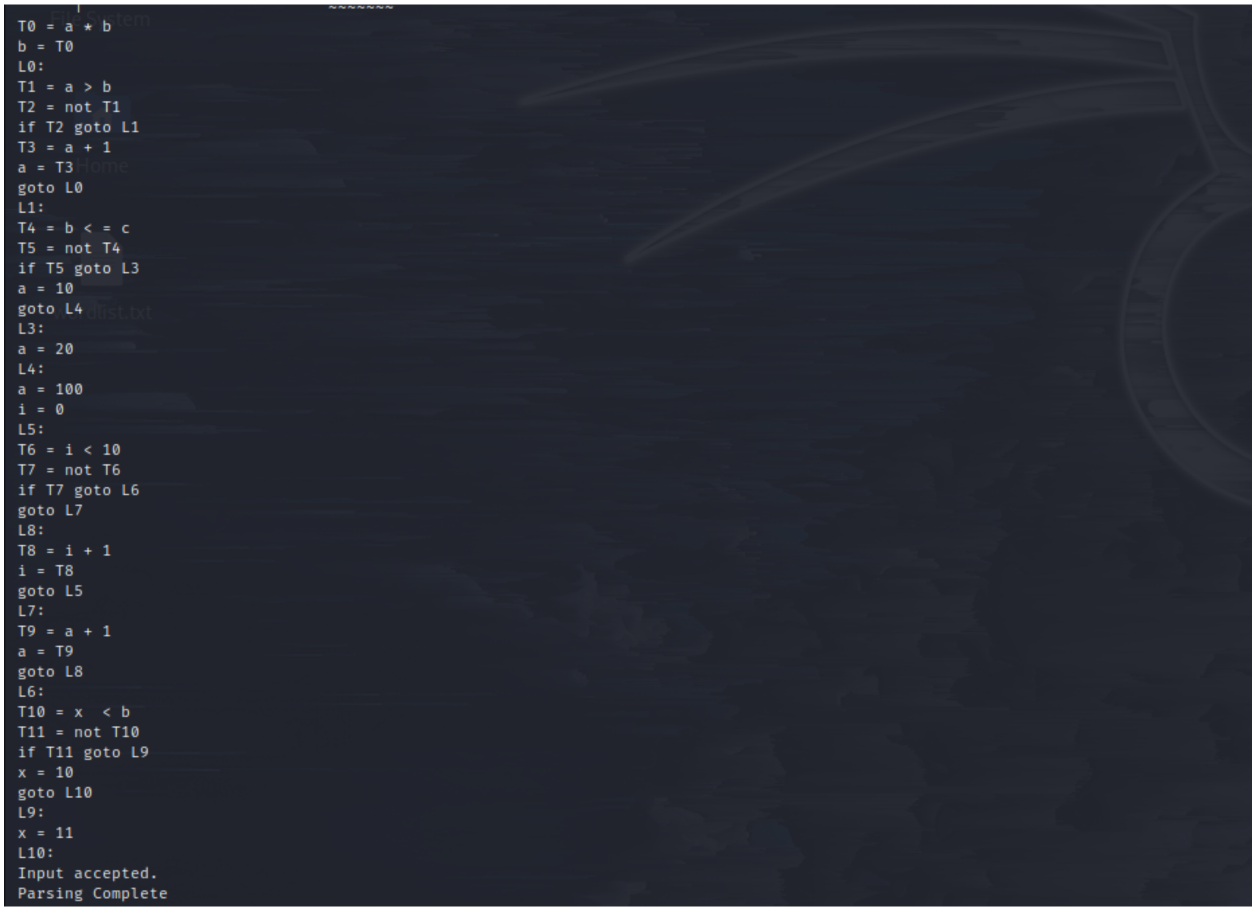
```
success
t0 = 2 + 3
a = t0
local.a = 5
t1 = 10 / a
t2 = a + t1
b = t2
 L_case1_1  : ifFalse a = 1 goto L_case2_1
t3 = a + 1
a = t3
 goto L_case0_0
   L_case2_1  : ifFalse a = 2 goto L_case0_0
 t4 = a - 1
a = t4
 goto L_case0_0
   L_case3_1 :t5 = a + 1
a = t5
 goto L_case0_0
 L_case0_0 : L1 :t6 = a > 0
ifFalse t6  goto L3
goto L2
 L2 :
t7 = a - 1
a = t7
 goto L1
L3 :
```

4. Abstract Syntax Tree (AST)



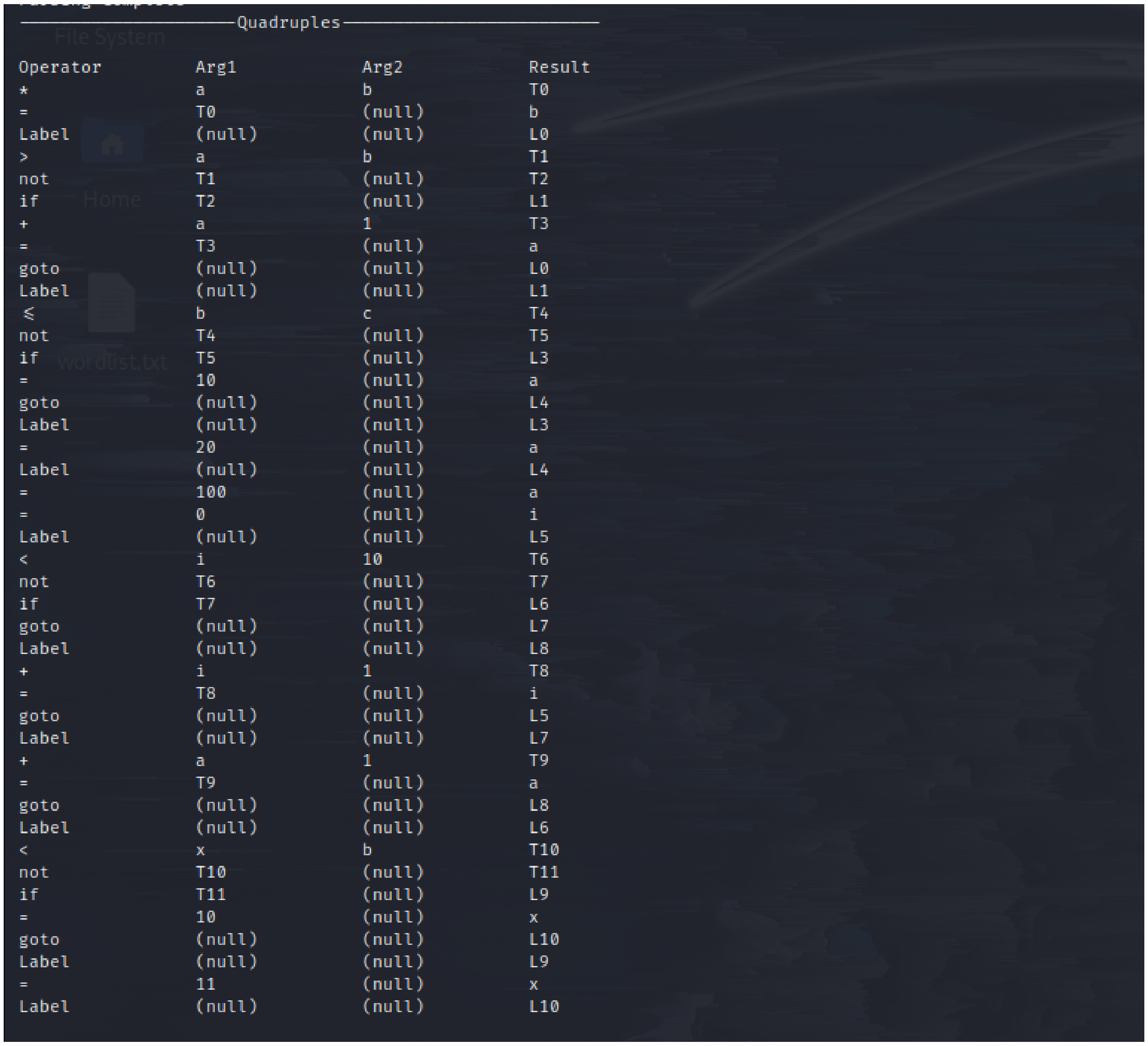
```
Graph 1:

Successful
```

5. Three Address Code Generation



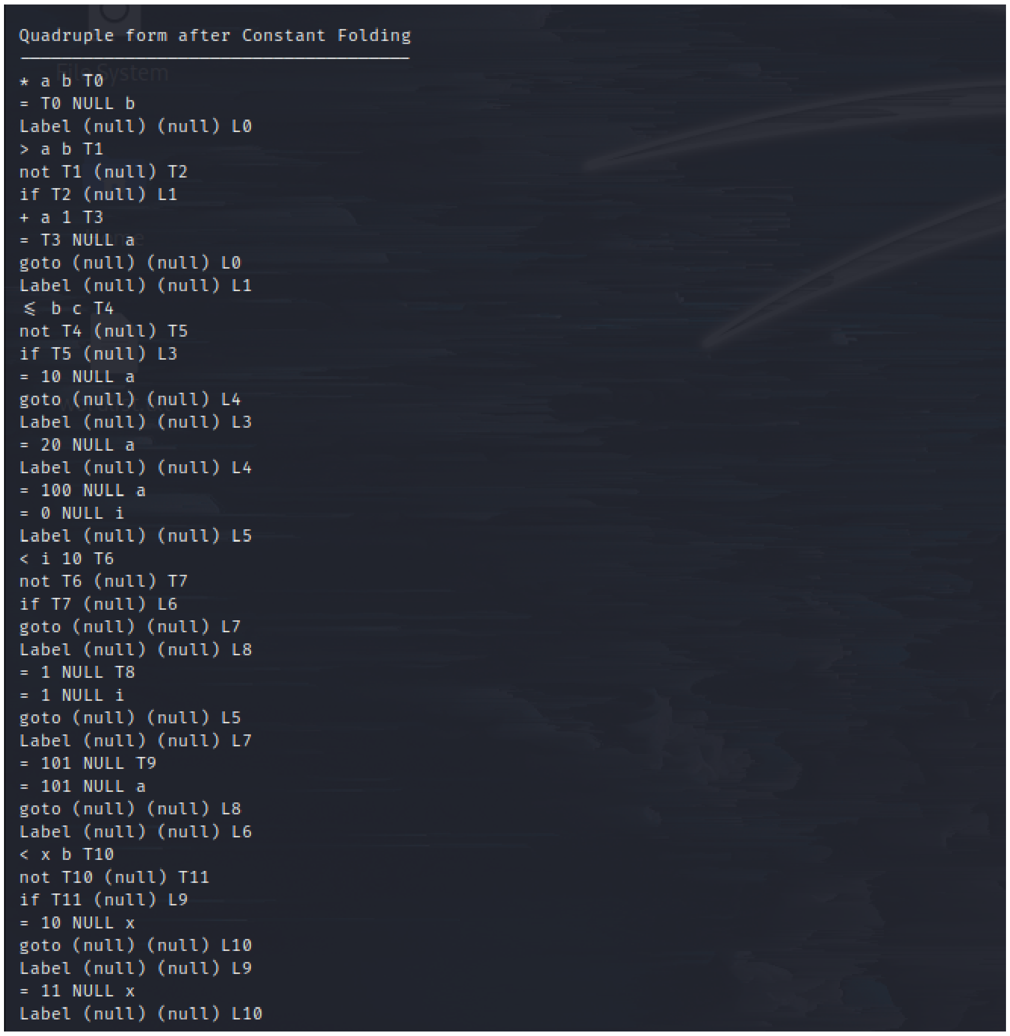
```
T0 = a * b
b = T0
L0:
T1 = a > b
T2 = not T1
if T2 goto L1
T3 = a + 1
a = T3
goto L0
L1:
T4 = b < = c
T5 = not T4
if T5 goto L3
a = 10
goto L4
L3:
a = 20
L4:
a = 100
i = 0
L5:
T6 = i < 10
T7 = not T6
if T7 goto L6
goto L7
L8:
T8 = i + 1
i = T8
goto L5
L7:
T9 = a + 1
a = T9
goto L8
L6:
T10 = x  < b
T11 = not T10
if T11 goto L9
x = 10
goto L10
L9:
x = 11
L10:
Input accepted.
Parsing Complete
```

6. Quadruple Generation



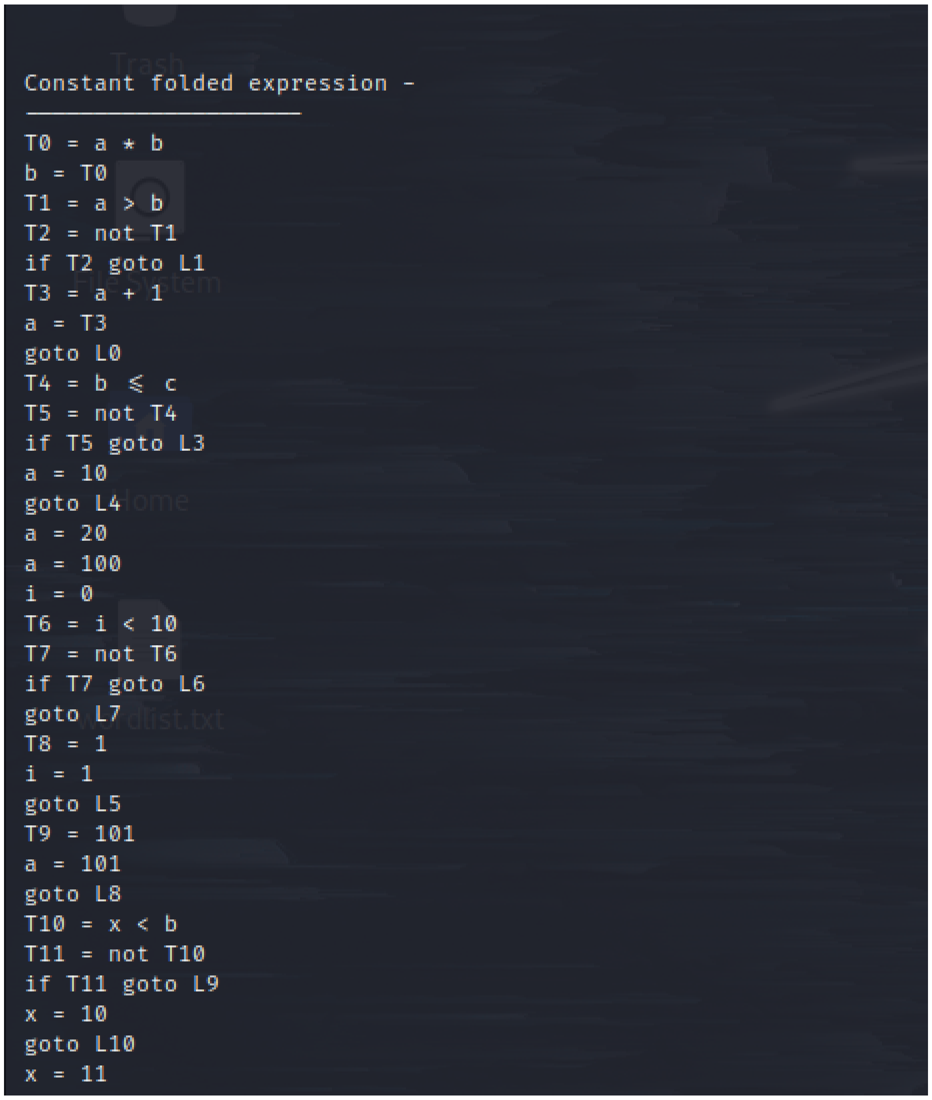
```
────────────────────Quadruples────────────────────────

Operator         Arg1            Arg2            Result
*                a               b               T0
=                T0              (null)          b
Label            (null)          (null)          L0
>                a               b               T1
not              T1              (null)          T2
if               T2              (null)          L1
+                a               1               T3
=                T3              (null)          a
goto             (null)          (null)          L0
Label            (null)          (null)          L1
⩽                b               c               T4
not              T4              (null)          T5
if               T5              (null)          L3
=                10              (null)          a
goto             (null)          (null)          L4
Label            (null)          (null)          L3
=                20              (null)          a
Label            (null)          (null)          L4
=                100             (null)          a
=                0               (null)          i
Label            (null)          (null)          L5
<                i               10              T6
not              T6              (null)          T7
if               T7              (null)          L6
goto             (null)          (null)          L7
Label            (null)          (null)          L8
+                i               1               T8
=                T8              (null)          i
goto             (null)          (null)          L5
Label            (null)          (null)          L7
+                a               1               T9
=                T9              (null)          a
goto             (null)          (null)          L8
Label            (null)          (null)          L6
<                x               b               T10
not              T10             (null)          T11
if               T11             (null)          L9
=                10              (null)          x
goto             (null)          (null)          L10
Label            (null)          (null)          L9
=                11              (null)          x
Label            (null)          (null)          L10
```

7. Quadruple Generation after Constant Folding



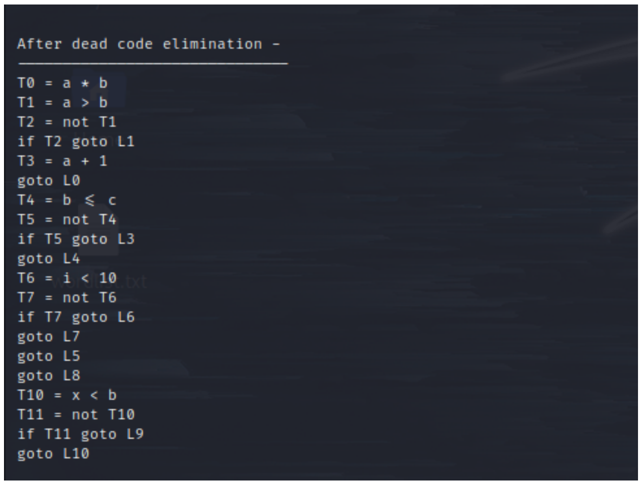
```
Quadruple form after Constant Folding
-------------------------------------
* a b T0
= T0 NULL b
Label (null) (null) L0
> a b T1
not T1 (null) T2
if T2 (null) L1
+ a 1 T3
= T3 NULL a
goto (null) (null) L0
Label (null) (null) L1
≤ b c T4
not T4 (null) T5
if T5 (null) L3
= 10 NULL a
goto (null) (null) L4
Label (null) (null) L3
= 20 NULL a
Label (null) (null) L4
= 100 NULL a
= 0 NULL i
Label (null) (null) L5
< i 10 T6
not T6 (null) T7
if T7 (null) L6
goto (null) (null) L7
Label (null) (null) L8
= 1 NULL T8
= 1 NULL i
goto (null) (null) L5
Label (null) (null) L7
= 101 NULL T9
= 101 NULL a
goto (null) (null) L8
Label (null) (null) L6
< x b T10
not T10 (null) T11
if T11 (null) L9
= 10 NULL x
goto (null) (null) L10
Label (null) (null) L9
= 11 NULL x
Label (null) (null) L10
```

8. **Constant Folded Expression**



```
Constant folded expression -
______________________

T0 = a * b
b = T0
T1 = a > b
T2 = not T1
if T2 goto L1
T3 = a + 1
a = T3
goto L0
T4 = b ≤ c
T5 = not T4
if T5 goto L3
a = 10
goto L4
a = 20
a = 100
i = 0
T6 = i < 10
T7 = not T6
if T7 goto L6
goto L7
T8 = 1
i = 1
goto L5
T9 = 101
a = 101
goto L8
T10 = x < b
T11 = not T10
if T11 goto L9
x = 10
goto L10
x = 11
```

9. **Dead Code Elimination**



```
After dead code elimination -
─────────────────────────────
T0 = a * b
T1 = a > b
T2 = not T1
if T2 goto L1
T3 = a + 1
goto L0
T4 = b ≤ c
T5 = not T4
if T5 goto L3
goto L4
T6 = i < 10
T7 = not T6
if T7 goto L6
goto L7
goto L5
goto L8
T10 = x < b
T11 = not T10
if T11 goto L9
goto L10
```