BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI
HYDERABAD CAMPUS
SECOND SEMESTER 2017 – 2018

COMPILER CONSTRUCTION (CS F363)

Home Assignment

Introduction

You will implement a compiler for a miniature programming language, too small a language that you can give a name to it. This language will help getting hands on practice to those concepts you learn in this course. This language must have a set of sequential statements, a conditional construct, loop construct, functions and arrays.

The compiler is divided into a few phases and implemented in stages.

Notes on implementation

Implementation of the project "must" be done in C. This is to ensure that all data structures and algorithms are hand-coded without the use of high level libraries and implementation must run on Linux.

Assignment Administration

- Project may be worked in teams of four. Choose your own team but you will not be allowed to change your team-mate later.
- Register your team details along with the language you wish to implement with Mr. Gourish OR Mrs. Meghna (PhD scholars) who sit in CSIS Research Scholars Lab, on or before 23/03/2018.
- Final Evaluation will happen between 15th and 25th April. Final evaluation comprises of Demo and viva-voce.
- Marking will be based not only on the implementation but also on your understanding of the implementation and the ability to explain your code and answer questions on your part of the work (For each phase split up of marks is mentioned).

Fair Practice

- Teams are permitted to discuss the project with each other but not allowed to see nor use each other's solutions.
- Plagiarism in any form is unacceptable. Project submissions will be rigorously scrutinized for plagiarism and the team members will be questioned to verify the ownership of the solution.

Phase-1 : Scanner Implementation (Marks 7)

In this assignment you write a DFA-based Lexical Analyser that recognizes some of the basic lexemes. Design, write, and thoroughly test the Language constructs. Write a driver program(parser) that calls your Scanner repeatedly, returning each token found by the Scanner in the input stream.

- Requirements Specification:
    - Input: Program File (example shown at the end)
    - Output: Return tokens either in the form of some number or as TK-identifier (example shown at the end)
    - Side Effects: White spaces removed
    - Exceptions: Invalid tokens

  Our language reserves all the key words that can appear in the language.

Scanner Deliverables

- **C Code** for scanner.
- Test cases and output files for test cases

Scanner sample Test Suite

Formulate your own test set / programs from the token list given as examples. The test set to be used for evaluation

Token List

- **Keywords**: int, float, boolean, string, while, until, if else, true, false
- **Operators**: +, -, *, /, %, :=, ==, >, <, >=, <=, !=, &&, ||, !, ?, :
- **Delimiters**: {, }, (,), [,], ;, ,
- **Identifiers**: must start with a letter (upper or lower case), and may contain zero or more additional characters as long as they are letters, digits, or underscores
- **Integer Literals**: may begin with an optional plus or minus followed by a sequence of one or more digits, provided that the first digit can only be zero for the number zero (which should not have a plus or minus before it).
- **Floating Point Literals**: may begin with an optional plus or minus followed by a sequence of one or more digits with the same provision above for integers, followed by a decimal point and one or more digits after the decimal point.
- **String literals**: start and end with a double quote followed by zero or more characters that may not be newlines, carriage returns, double quotes, or backslashes. The only exceptions are reserved escape sequences which are limited to the following: \t, \n, \r, \", and \\.

Phase-1 How-to

1. Read the language Specification: the overview, the grammar (natural form) and the tokens to gain an over-all understanding.
2. Apply your understanding to write Regular Expression and convert them into NFA.
3. Convert NFA into DFA.
4. Implement the DFA.
5. Test your Scanner with the given test cases.
6. Write your own test cases and document your code
7. **To make your DFA as small as possible and manage the code you may store all reserved words in an string array and first check if the input is matching with any string in the array you can declared it as keyword otherwise you search in your DFA.**
8. **For a smooth transition into the next phase first write the Context Free Grammar / BNF for the mini language and from the grammar identify the terminals which can be grouped into tokens. Draw DFA for each tokens individually and then combine followed by implementing it.**

Your code will be evaluated for 7 marks on the following

1. Identifying the tokens – 1.5
2. Eliminate comments and while spaces. – 1.5
3. Generate errors - 1
4. Keep track of line numbers – 1
5. Viva – 2

If I run this on the file

```
/* A program to compute factorials */


int fact( int n) {
if (n <= 1)
return 1;
else
return n*fact(n-1);
}
void main(void) {
int x;
x = 1;
while (x <= 10) {
write(x);
write(fact(x));
writeln();
x = x + 1;
}
}
```

I get the following token stream:

```
Token 100, string int, line number 3
Token 132, string fact, line number 3
Token 128, string (, line number 3
Token 100, string int, line number 3
Token 132, string n, line number 3
Token 129, string ), line number 3
Token 114, string {, line number 3
Token 103, string if, line number 4
Token 128, string (, line number 4
Token 132, string n, line number 4
Token 123, string <=, line number 4
Token 130, string 1, line number 4
Token 129, string ), line number 4
Token 106, string return, line number 5
Token 130, string 1, line number 5
Token 110, string ;, line number 5
Token 104, string else, line number 6
Token 106, string return, line number 7
Token 132, string n, line number 7
Token 118, string *, line number 7
Token 132, string fact, line number 7
Token 128, string (, line number 7
Token 132, string n, line number 7
Token 117, string -, line number 7
Token 130, string 1, line number 7
```
and so forth. Your token kinds, being constants, are likely to be different from mine, but your string values and line numbers should be the same.

Phase-2 : Parser Construction **(Marks 7)**

• Requirements Specification:

o Input: Lexer generated tokens
o Output: Parse Tree in the form of any tree traversal or level wise output of the nonterminal from left to right. In case of errors your parser must report the errors and continue parsing.

• Files
o Interface file : parser.h
o Implementation : parser.c

Phase-2: How-to

1. Read the language Specification: the overview, the grammar (natural form) and the tokens to gain an over-all understanding.
2. Apply your understanding to the given examples and work out the details by deriving the program text using the rules given in the grammar.
3. Understand the any type of parser for your grammar.
4. Implement the paser using techniques discussed in the classroom
5. Test the scanner with the test cases.
6. Write a procedure to traverse and print the parse tree.
7. Write your own test cases and document your code.

Your code will be evaluated for 8 marks on the following

6. Parse table by hand – 2
7. Parsing algorithm and printing the parse tree or stack contents – 1
8. On error the parser must not stop but continue -1
9. A few Error handling routines -1
10. Viva – 2

Phase-3 : Three Address code Generation **(Marks 6)**

In this third part of the compiler project, you will be extending the compiler developed for the language you have chosento generate intermediate code using the three-address instruction described in class. In essence you will write a C program and translate it into low-level three-address code using quadruples. As discussed in the class you will treat all the types of instructions differently. Code generation for if, loops and functions will be covered in the class in detail.

For example, if the input program in c language is as follows:

```
#include<stdio.h>
  int main(){
  int n,fact=1;
  printf("enter the number");
  scanf("%d",&n);
  for(i=1;i<n;i++)
    fact*=i;
  printf("factorial = %d",fact);
  return 0;
}
```

The output three address code will be

```
func begin main
        fact=1
        refparam “enter the string”
        call printf,2
        refparam “%d”
        refparam n
        call scanf,3
        i=1
L1: if i>n goto L2
        T1=fact*n
        Fact=T1
        T2=i+1
        i=T2
        goto L1
L2:  refparam "factorial = %d"
        refparam fact
        call printf,3
        return 0
func end
```

Your code will be evaluated for 6 marks on the following

1. Simple expressions and assignment statements -1
2. Relations expressions and Boolean operators - 1
3. If statement -1
4. Loops – 1
5. Functions -1
6. Viva – 1

I/C CSF 363