**Compiler Design for subset of C language**

**The Project: overview**

* The project consists of-
  + lexical analysis
  + Syntax analysis (parsing and generation of syntax tree)
  + Semantic analysis
  + Generation of intermediate code
  + Code generation

**Goals and motivation:**

* A complete compiler from source code to machine code

Motivation:

* Understand all parts of a compiler
* Understand the execution of a program
* Experience of a larger programming project.

**Implementation language:**

Subset of C programming language (common feature of the C language)

**Required Tools:**

1. Scanner: flex
2. Parser: bison

**TOOL DESCRIPTION:**

## **Flex Program:**

Flex is a tool for generating scanners: programs which recognized lexical patterns in text. Flex reads the given input files, or its standard input if no file names are given, for a description of a scanner to generate. The description is in the form of pairs of regular expressions and C code, called rules. Flex generates as output a C source file, `lex.yy.c', which defines a routine `yylex ()'. This file is compiled and linked with the `-lfl' library to produce an executable. When the executable is run, it analyzes its input for occurrences of the regular expressions. Whenever it finds one, it executes the corresponding C code.

## **Bison program:**

## Bison is a general-purpose parser generator that converts an annotated context-free grammar into a deterministic LR or generalized LR (GLR) parser employing LALR(1) parser tables. As an experimental feature, Bison can also generate IELR(1) or canonical LR(1) parser tables. Bison is upward compatible with Yacc: all properly-written Yacc grammars ought to work with Bison with no change.

Input files should follow the yacc convention of ending in .y. Unlike yacc, the generated files do not have fixed names, but instead use the prefix of the input file. For instance, a grammar description file named parse.y would produce the generated parser in a file named parse.tab.c, nstead of yacc's y.tab.c.

**Language definition:**

**Included in Sub set of C-**

* datatypes: int, char, arrays
* functions, parameter passing
* if-statements, while loops
* expresions: constants, identifiiers, function calls, array indexing
* literals: characters and integers
* unary expressions: - !
* binary operators: + - \* / < > <= >= == != &&

**Not included in Sub set of C-**

* Multidimensional arrays, pointers, structures
* floating point numbers
* initialization in declarations
* functions that take a variable number of arguments
* declarations in nested blocks

**Lexical elements**

* Decimal integer constants and character literals. A character literal contains either a single printable character, or the \n escape sequence (line break). A character literal denotes an integer constant whose value is the representation code of the character.
* Alphanumeric identifiers: non-empty sequences of letters or digits starting with a letter. An underscore is treated as a letter.
* Keywords: char, else, if, int, return, void, and while.
* Special symbols: !=, !, &&, (, ), \*, +, , (comma), -, /, ;, <=, <, ==, =, >=, >, [, ], {, }.
* White space characters: blank (32), newline (10), carriage return (13), form feed (12), and tab (9). The numbers are the ASCII representation codes for the characters.

Comments: /\* followed by anything and terminated by \*/, and // followed by anything and terminated by end of line.

**Syntax**

* Primary expressions: constants, identifiers, function calls, array indexing, and expressions within parentheses.

Unary expressions with the ! and - unary operators.

Binary expressions with the +, -, \*, /, <, >, <=, >=, ==, !=, &&, and = operators.

Statements: expression statements, the empty statement, if statements with or without else, while statements, return statements, and compound statements (blocks), i.e., statements enclosed in { }.

Local variable declarations are only permitted at the top-level function body block, not in nested blocks.

* Variable declarations: base type followed by variable name, and for arrays followed by the array size (an integer constant) in square brackets.

Multi-dimensional arrays, pointers, and structures are not included.

Initializes in variable declarations are not included.

* Function definitions: return type or void, function name, parameter list, and body (compound statement) in that order.

The parameter list is either void, meaning no parameters, or a sequence of variable declarations separated by, (comma). An array parameter in a function head is written without array size, i.e., with only the brackets.

An external (library) function can be declared by writing a function definition without body, terminated with a ; (semi-colon).

**Program execution**

* Execution starts at the user-defined function main which takes no parameters and returns int. Execution ends when main returns.

**Organization of the project**

* lexical analysis
* parser
* semantic analysis
* generation of intermediate code
* generation of machine code

**Lexical analysis**

Input: A C source file

Output: A "stream" of tokens, errors

**Parser (Syntax analysis)**

Input: A stream of tokens

Output: Syntax tree, errors

**Semantic analysis**

Input: Syntax tree

Output: errors (or a syntax tree that is known to be semantically correct)

**Generation of intermediate code**

Input: Syntax tree (known to be correct)

Output: Intermediate code (some form of register transfer language)

**Generation of machine code (MIPS)**

Input: Intermediate code

Output: A file containing MIPS assembly code

**References:**

[**https://gnuu.org/2009/09/18/writing-your-own-toy-compiler/**](https://gnuu.org/2009/09/18/writing-your-own-toy-compiler/)

[**https://norasandler.com/2017/11/29/Write-a-Compiler.html**](https://norasandler.com/2017/11/29/Write-a-Compiler.html)