***COMPILER***

***Introduction***

A **compiler** is a program that translates another program from source language (evolved programming language) in object language (machine code or another evolved language for which there is a compiler). During translation, the compiler displays some error messages, and can even be able to correct some errors.

Source program

Compiler

Object program

Error messages

The compilation is divided into two stages, which are divided into sub-stages:

1. **Analysis stage**: it aims to divide the source program into pieces that can be completely defined. The analysis phase in the compilation deals with the modification of the source program, which is done taking into account the type of grammatical rules of the source language given as input.

This stage is divided in three types of analysis:

* **Lexical analysis**
* **Semantic analysis**
* **Syntactic analysis.**

1. **Synthesis stage:** it deals with the generation of the intermediate code and the final generation of the object program.

This stage is divided in three types of synthesis:

* **The generator of the intermediate code**
* **The optimization of the code**
* **The generator of the code.**

**Representative schema for the compilation stages**

Lexical analyzer(scanner)

Syntactic analyzer(parser)

Semantic analyzer

The generator of the intermediate code

The optimization of the code

The generator of the code.

Symbol table manager

Error recovery

Synthesis

Analysis

Source program

Object program

Inaddition to the phases presented above, there is also the phase of the **symbol table manager** and the **error recovery** phase.

One of the essential functions of a compiler is to record the identifiers used in a source program, but also to gather information about different attributes of each identifier found.

A **symbol table** is a data structure that contains a record for each field identifier, for each identifier attribute. Such a data structure must be organized in such a way as to allow the storage, but also the quick retrieval of information (we suggest the Hash-tables).

When an identifier is detected in the analysis phase, it is immediately introduced in symbol analysis. However, the attributes of the identifiers cannot all be defined at this stage, following which in the next phases the information about identifiers will be introduced.

As an example: During the sematic analysis, but also when we are generating the intermediate code, it is necessary to know what kind of identifiers are so we can verify if the source program is using them correctly.

Now, let’s take each phase of the compiler in more detail:

1. **Analysis phase:**
   1. **Lexical analysis:**

In the lexical analysis phase all the characters are grouped into tokens. All characters that make up a token are called lexeme for token. Each token is associated with a certain lexical value.

Example:

value := initialValue + cant \* 40 => id1 := id2 + id3 \* 40

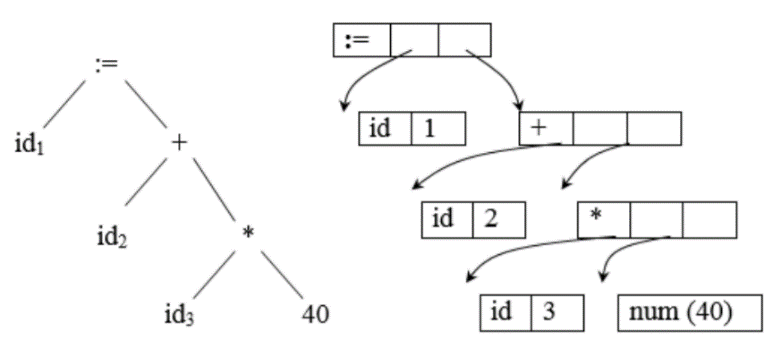
Obs!

* You can also define a token for “:=”, but it would make the exposure itself difficult. Also, a token can be added for the constant value 40.
* In the symbol table “value” will be introduced at the moment of lexical detection of id1 and analogous to the others.
* The rule that describes the set of lexemes that form a lexical unit is called pattern.
  1. **Syntactic analysis:**

It is the second stage of the analysis phase and in this stage we try to construct a derivation tree according to a set of predefined rules.

Obs!

* In the lexical analysis phase the tree leaves are solved most often using grammars and machines.
* In the syntactic analysis phase, the main nodes of the derivation tree are defined.



This is how the derivation tree looks for the above example

* 1. **Semantic analysis:**

After passing from the syntactic analysis both, the logical interaction between tokens and the verification of their types will be analyzed. The most important step is to check the types; thus the semantic analyzer verifies that each operand with which an operator operates is the type required in the programming language.

1. **Synthesis phase:**

It is the second main stage of the compilation and is divided in:

* 1. **The generator of the intermediate code**
  2. **The optimization of the code**
  3. **The generator of the object code.**

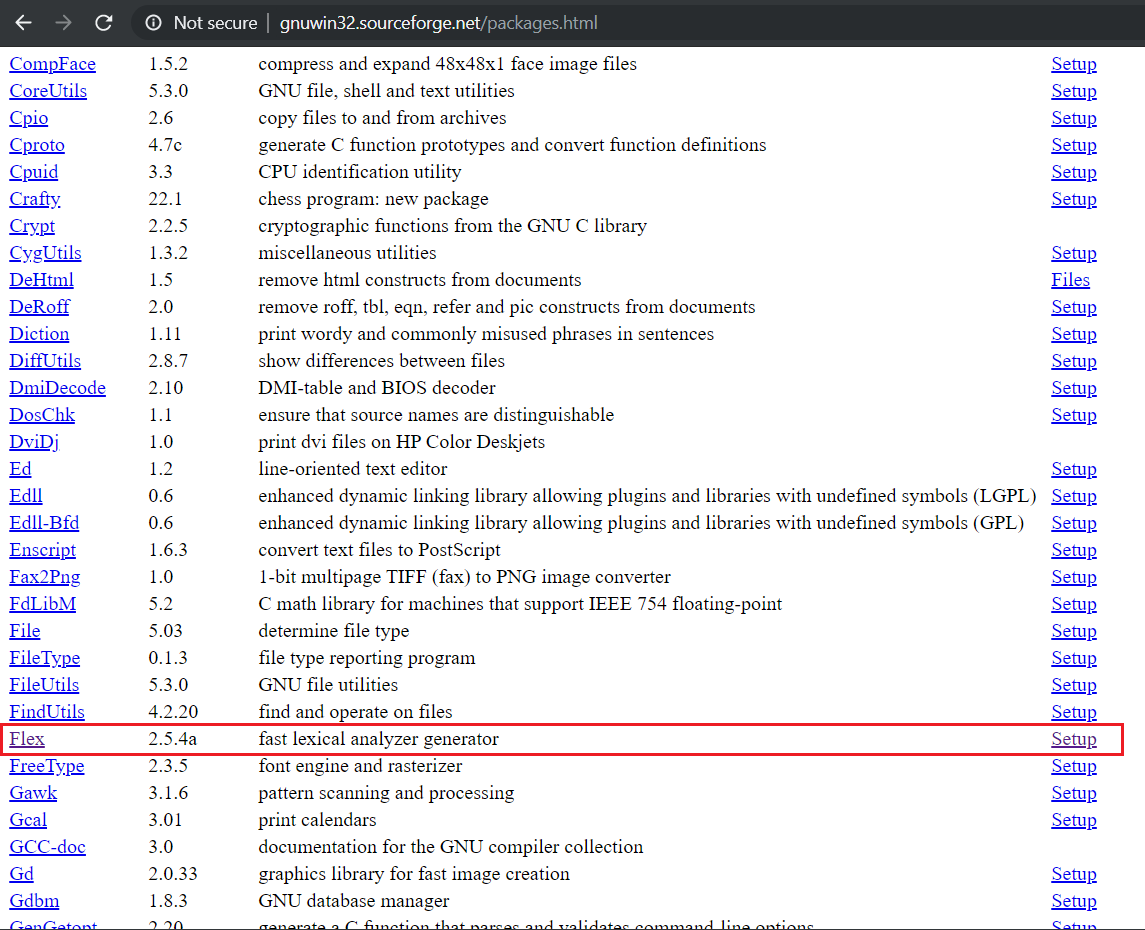
Depending on how much detail we want to exist in the synthesis phase, the first two sub-stages may be missing, but the one for generating the object code represents the essence of the synthesis phase. This step consists in generating an assembly code in which memory locations are selected for each of the variables used and each translated into an assembly code sequence.

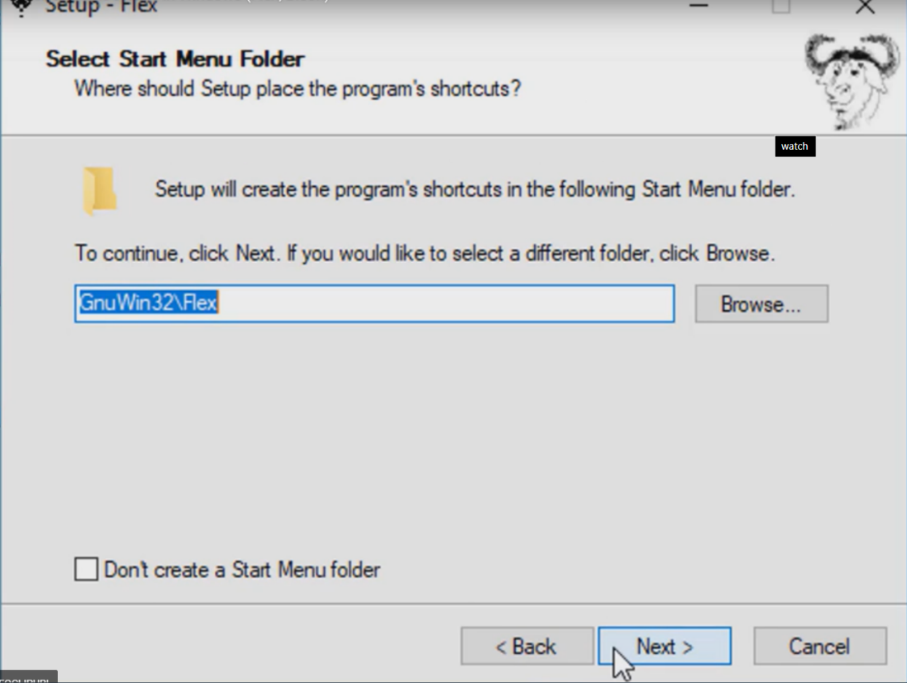
Following the two phases of analysis and synthesis, various errors may occur or they may be discovered. However, after finding errors, one phase must continue to detect other errors without the source program, so it must repair the error in such a way that the compilation process can continue.

***Setup the tools for the compiler***

***Setup for Flex***:

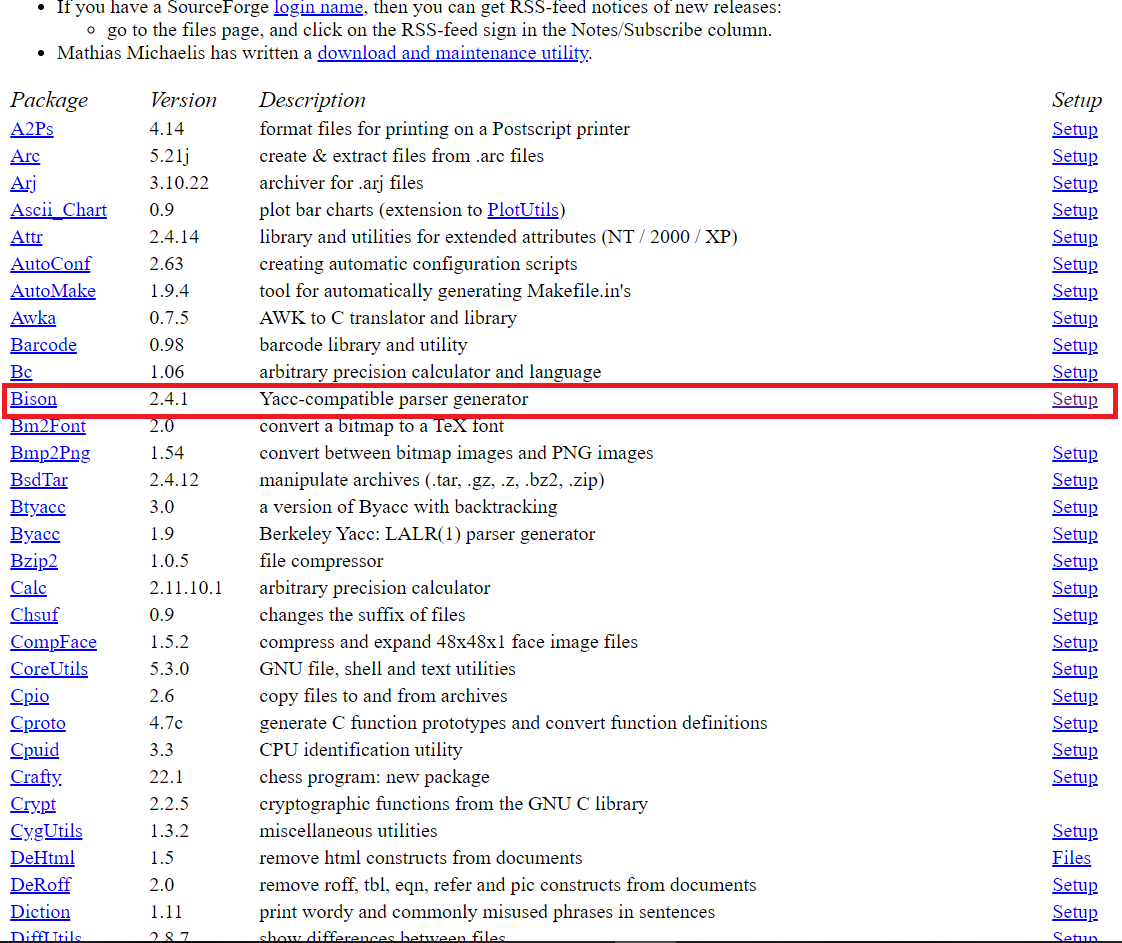
Here you can find the package for FLEX: <http://gnuwin32.sourceforge.net/packages.html>



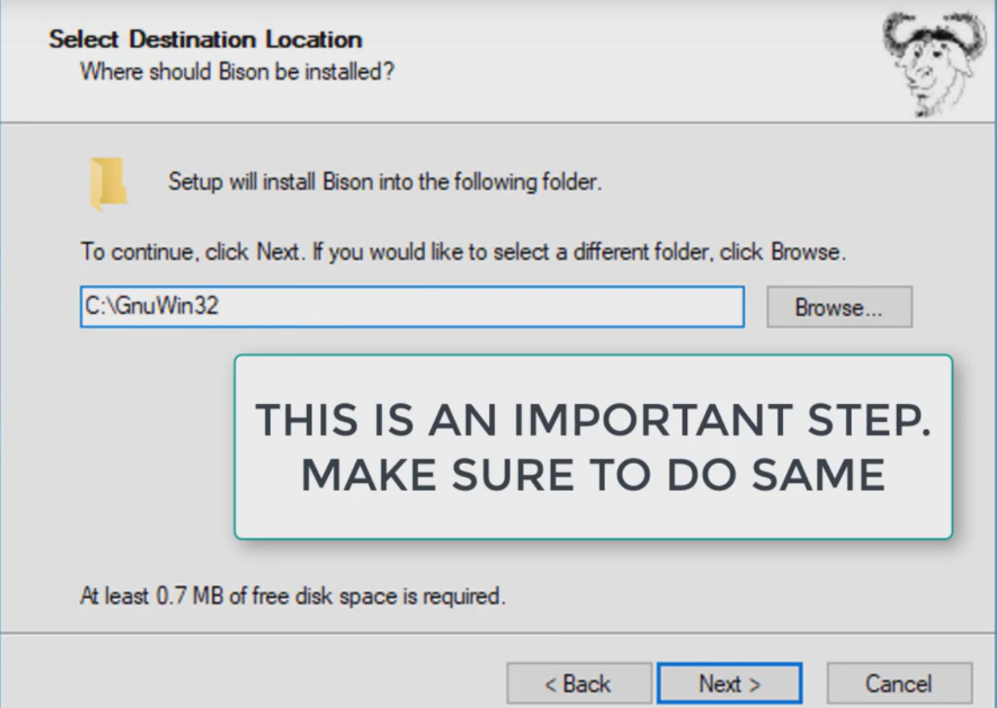
**!!!** After downloading, when you install the flex package, please make sure to follow this important step: save the package in a folder named GnuWin32, not in the folder of program files.

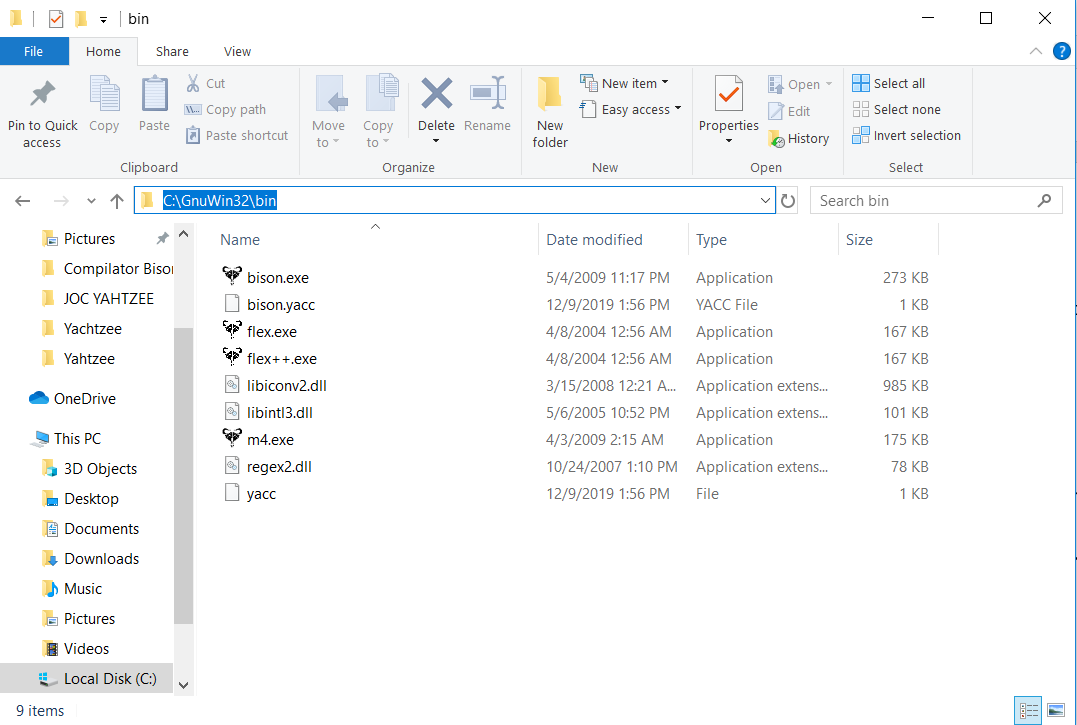
***Setup for Bison***:

Here you can find the package for Bison: <http://gnuwin32.sourceforge.net/packages.html>



**!!!** After downloading, when you install the bison package, please make sure to follow this important step: save the package in a folder named GnuWin32, not in the folder of program files.

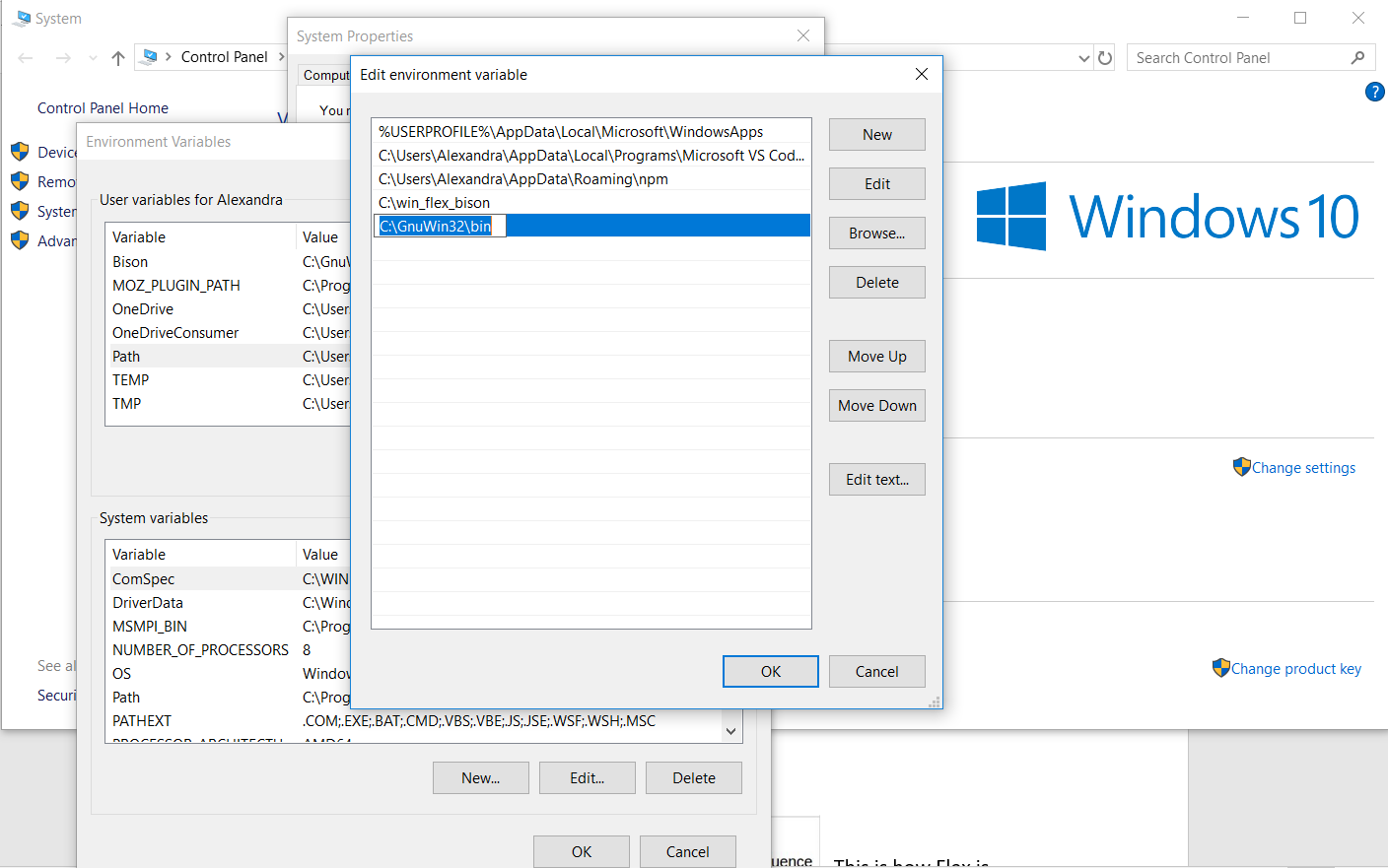




After installing, go in the GnuWin32 folder, in the bin folder and copy the path.

!!! Now we need to add the binaries to path so we can run them on cmd.exe.

After you’ve copied the path, go in the properties of this pc, in “Advanced system settings”, then go to “Environment variables”, select from the list the “Path” option and the press edit. Add a new line where you paste the path copied from the file explorer.

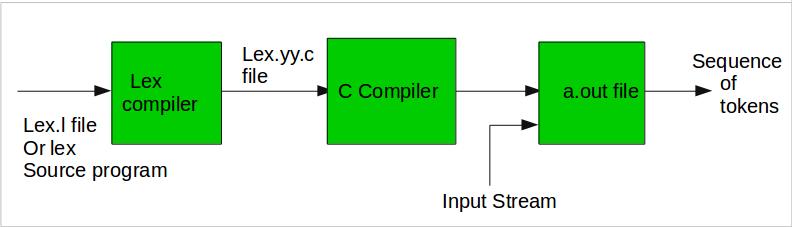


Then save the changes and this is the setup for the tools we are going to use for our compiler.

***Lexical Analysis***

**FLEX (Fast Lexical Analyzer Generator)** is a tool/computer program for generating lexical analyzers (scanners or lexers) written by Vern Paxson in C around 1987. It is used together with Berkeley Yacc parser generator or **GNU Bison parser generator**.

***Working with FLEX***:



This is how Flex is working for your compiler.

**Step 1:** An input file describes the lexical analyzer to be generated named lex.l is written in lex language. The lex compiler transforms lex.l to C program, in a file that is always named lex.yy.c.  
**Step 2:** The C compiler compile lex.yy.c file into an executable file called a.out.  
**Step 3:** The output file a.out take a stream of input characters and produce a stream of tokens.

***Program structure***:

1. **Definition Section:** The definition section contains the declaration of variables, regular definitions, manifest constants. In the definition section, text is enclosed in **“%{ %}”** brackets. Anything written in this brackets is copied directly to the file **lex.yy.c**

Syntax:

%{

// Definitions

%}

In our compiler we have here the declarations and the includes.

Example:

%{

#include <stdlib.h>

#include <stdio.h>

#include "keyword.h"

void yyerror(char \*);

keyword\_t result;

int yylineno;

%}

1. **Rules Section:** The rules section contains a series of rules in the form: *pattern action* and pattern must be unintended and action begin on the same line in {} brackets. The rule section is enclosed in **“%% %%”**.

Syntax:

%%

pattern action

%%

Here you need to implement some regular expressions to find the comments, the symbols, the predefined words, the declarations.

|  |  |
| --- | --- |
| ***Pattern*** | ***The matches*** |
| (0|[1-9][0-9]\*) | This expression recognizes the integers. |
| [-()<>=+\*/;{}\%] | This expression recognizes the symbols for the operations. |
| “>=” | This expression recognizes the symbols of greater or equal so later you can use it for making the comparations. |
| “<=” | This expression recognizes the symbols of less or equal so later you can use it for making the comparations. |
| “==” | This expression recognizes the symbols of equal so later you can use it for making the comparations. |
| “!=” | This expression recognizes the symbols of not equal so later you can use it for making the comparations. |
| [a-zA-Z][\_a-zA-Z0-9]\* | This expression recognizes the names of variables. |
| [ \t]+ | This expression finds the whitespaces and will ignore them. |
| \n | This expression finds the new lines. |
| . | In the end, if the expression doesn’t match any other pattern, it will return an error. |

Let’s take as example the part of the code where you recognize the numerical value:

Example:

(0|[1-9][0-9]\*) {

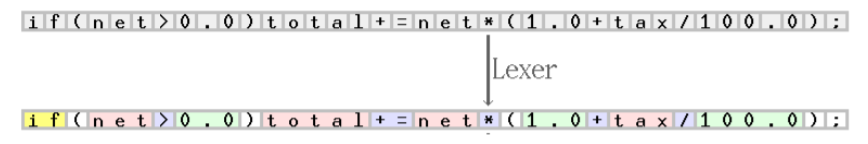
yylval.iValue = atoi(yytext);

return INTEGER;

}

First of all, it checks by using the regular expression if there is a number. If this condition is satisfied, we convert the string into an integer, we add this number in our variable of values and the we return “INTEGER” so the compiler will know that it has to deal with a number and it will treat it in the correct places.

By using this regular expressions you will tokenize the expression like this:



Condition

Name of the variable

The comparation

Name of the variable

The operation

1. **User Code Section:** This section contains C statements and additional functions. We can also compile these functions separately and load with the lexical analyzer.

In our compiler this section has only the following code:



The function yywrap() is called when the scanner encounters the end of the file(input). If yywrap() returns 0 then the scanner continues scanning but when yywrap() returns 1 this means that the end of the file has encountered.

***Syntax Analysis***

**GNU Bison**, commonly known as Bison, is a [parser generator](https://en.wikipedia.org/wiki/Parser_generator) that is part of the [GNU Project](https://en.wikipedia.org/wiki/GNU_Project). Bison reads a specification of a [context-free language](https://en.wikipedia.org/wiki/Context-free_language), warns about any [parsing](https://en.wikipedia.org/wiki/Parsing) ambiguities, and generates a parser (either in [C](https://en.wikipedia.org/wiki/C_(programming_language)), [C++](https://en.wikipedia.org/wiki/C%2B%2B), or [Java](https://en.wikipedia.org/wiki/Java_(programming_language))) which reads sequences of [tokens](https://en.wikipedia.org/wiki/Lexical_analysis#Token) and decides whether the sequence conforms to the syntax specified by the grammar. The generated parsers are portable: they do not require any specific compilers. Bison by default generates [LALR(1) parsers](https://en.wikipedia.org/wiki/LALR_parser) but it can also generate [canonical LR](https://en.wikipedia.org/wiki/Canonical_LR_parser), IELR(1) and [GLR](https://en.wikipedia.org/wiki/GLR_parser) parsers.

An **LALR parser** or **Look-Ahead LR parser** is a simplified version of a [canonical LR parser](https://en.wikipedia.org/wiki/Canonical_LR_parser), to parse (separate and analyze) a text according to a set of [production rules](https://en.wikipedia.org/wiki/Production_(computer_science)) specified by a [formal grammar](https://en.wikipedia.org/wiki/Formal_grammar) for a [computer language](https://en.wikipedia.org/wiki/Computer_language). ("LR" means left-to-right, [rightmost derivation](https://en.wikipedia.org/wiki/Rightmost_derivation).).

A **canonical LR parser** or **LR(1) parser** is an [LR(k)](https://en.wikipedia.org/wiki/LR(k)) parser for *k=1*, i.e. with a single [lookahead](https://en.wikipedia.org/wiki/Parsing#Lookahead) [terminal](https://en.wikipedia.org/wiki/Terminal_symbol). The special attribute of this parser is that any LR(k) grammar with *k>1* can be transformed into an LR(1) grammar. However, back-substitutions are required to reduce k and as back-substitutions increase, the grammar can quickly become large, repetitive and hard to understand. LR(k) can handle all [deterministic context-free languages](https://en.wikipedia.org/wiki/Deterministic_context-free_language).

A **GLR parser** (GLR standing for "Generalized LR", where L stands for "left-to-right" and R stands for "rightmost (derivation)") is an extension of an [LR parser](https://en.wikipedia.org/wiki/LR_parser) algorithm to handle [non-deterministic](https://en.wikipedia.org/wiki/Deterministic_context-free_grammar) and [ambiguous grammars](https://en.wikipedia.org/wiki/Ambiguous_grammar).

Simpler, Bison is a program that takes as input a specification of a syntax, and produces as output a procedure for recognizing that language. It’s input file is similar to that required by flex. Generally bison input files are given a .y or .yacc extension. Bison generates a parser function named yyparse(), which you can then call from a main program.

A bison input file consists of 3 sections: definitions, rules, and user subroutines. These sections are separated by two percent signs. The first 2 sections are required although one may be empty.

/\* definitions \*/

....

%%

/\* rules (grammar) \*/

....

%%

/\* user subroutines \*/

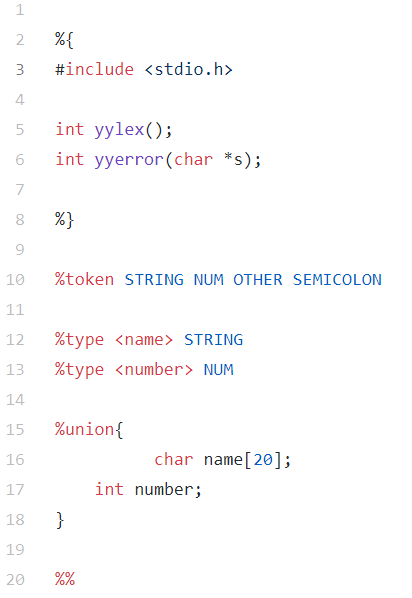
....

* The **definition section** includes information about the tokens used in the syntax definition. But, like flex, this section can also contain a literal block of C/C++ code which is copied to the beginning of the generated C file.
* The **rules section** is like that of flex. You specify a pattern for a production and the code, if any, that is executed when the rule is matched.
* The **user subroutines section** is copied to the C file. Normally the main is put here and have it call yyparse(), which is the name of the parsing function generated by bison.

Bellow is an example of such a bison input file:

**Definition section:**

Literal block of C/C++ code set off using %{ and %}

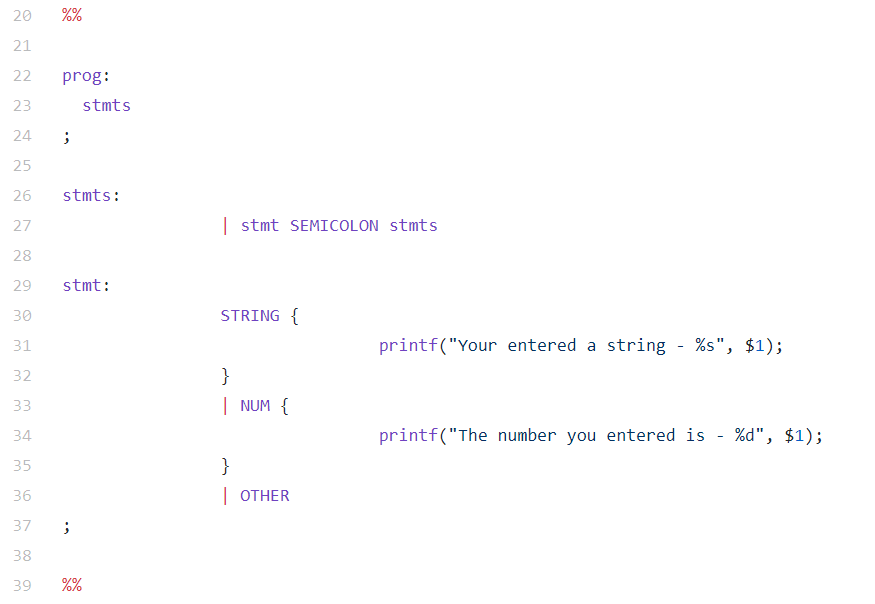


we need the **yylex** and **yyerror** prototypes because bison uses them in its generated code, but does not declare them for us automatically.

**(…)**

**%token** lines define symbols which represent the values which will be returned by the lexer and correspond to the terminals in your rules. All symbols used as tokens must be defined in this section although not all need be on the same line.

**Rules section:**

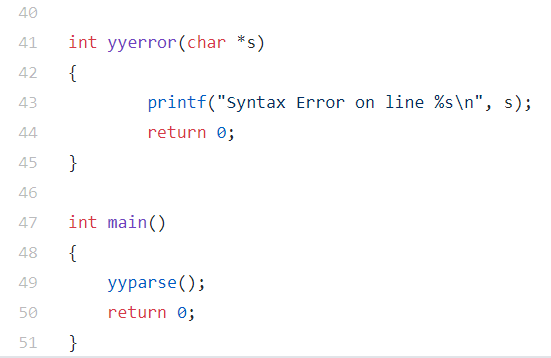


Explanation of the tokens and action to be made on token recognition.

**stmts: | …**Indicating that stmts token can also be null (lambda).

Entry-point (Can also be defined using **%start**)

**User subroutines section:**



User defined subroutines and **main function** calling **yyparse()**, the parsing function that bison generates.

Other declarations which can be contained in this section are:

**(…)**

* **%union**: defines the structure that the lexer will use to pass lexemes[[1]](#footnote-1) to the parser.
* **%start**:
* **%left, %right and %nonassoc**: declare operator associativity and precedence ( %left - specifies tokens which are left associative, %right - specifies tokens which are rigth associative, %nonassoc - tokens which are non-associative)
* **%type**: specifies names for non-terminals in your rules. These may or may not have a type assigned.

**Abstract syntax tree**

Abstract syntax tree or just syntax tree, is a tree representation of the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a construct occurring in the source code.

The syntax is "abstract" in the sense that it does not represent every detail appearing in the real syntax, but rather just the structural or content-related details. For instance, grouping parentheses are implicit in the tree structure, so these do not have to be represented as separate nodes. Likewise, a syntactic construct like an if-condition-then expression may be denoted by means of a single node with three branches.

This distinguishes abstract syntax trees from concrete syntax trees, traditionally designated parse trees. Parse trees are typically built by a parser during the source code translation and compiling process. Once built, additional information is added to the AST by means of subsequent processing, e.g., contextual analysis.

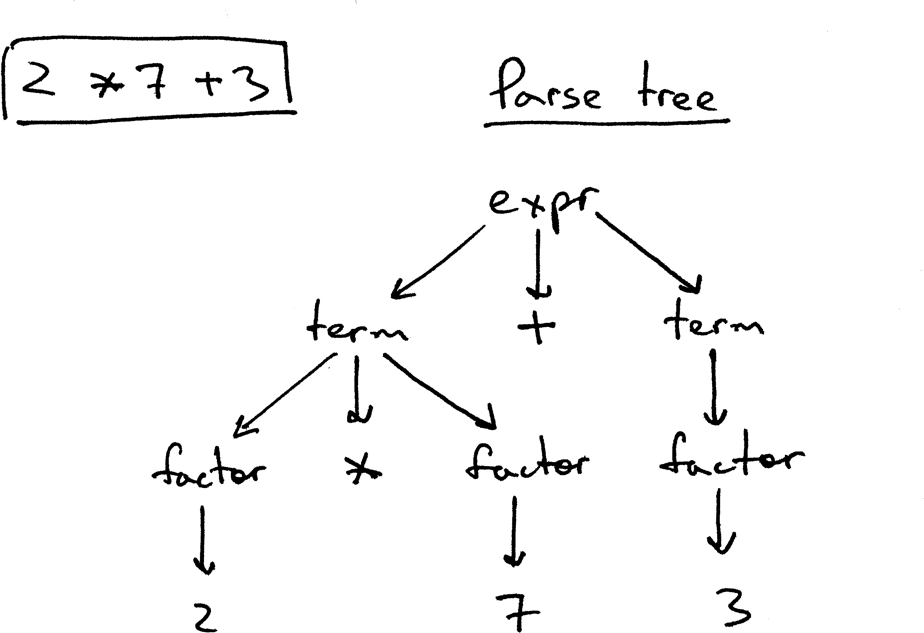
Abstract syntax trees are also used in program analysis and program transformation systems.

But before we dig deeper into ASTs let’s talk about parse trees briefly. Though we’re not going to use parse trees for our interpreter and compiler, they can help you understand how your parser interpreted the input by visualizing the execution trace of the parser. We’ll also compare them with ASTs to see why ASTs are better suited for intermediate representation than parse trees.

So, what is a parse tree? A parse-tree (sometimes called a concrete syntax tree) is a tree that represents the syntactic structure of a language construct according to our grammar definition. It basically shows how your parser recognized the language construct or, in other words, it shows how the start symbol of your grammar derives a certain string in the programming language.

The call stack of the parser implicitly represents a parse tree and it’s automatically built in memory by your parser as it is trying to recognize a certain language construct.

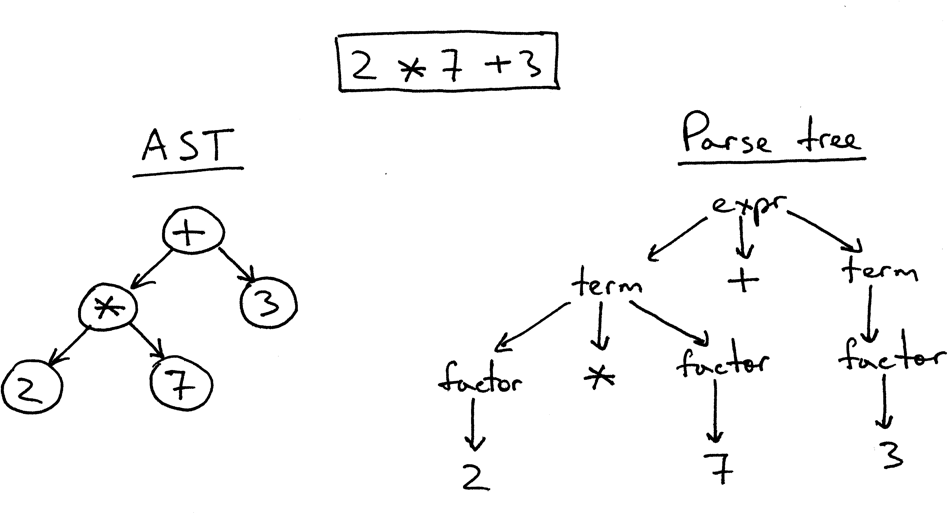
Let’s take a look at a parse tree for the expression 2 \* 7 + 3:



In the picture above you can see that:

* The parse tree records a sequence of rules the parser applies to recognize the input.
* The root of the parse tree is labeled with the grammar start symbol.
* Each interior node represents a non-terminal, that is it represents a grammar rule application, like *expr*, *term*, or *factor* in our case.
* Each leaf node represents a token.

Now lets take a look at both the AST and the parse tree for the expression 2 \* 7 + 3:



As you can see from the picture above, the AST captures the essence of the input while being smaller.

Here are the main differences between ASTs and Parse trees:

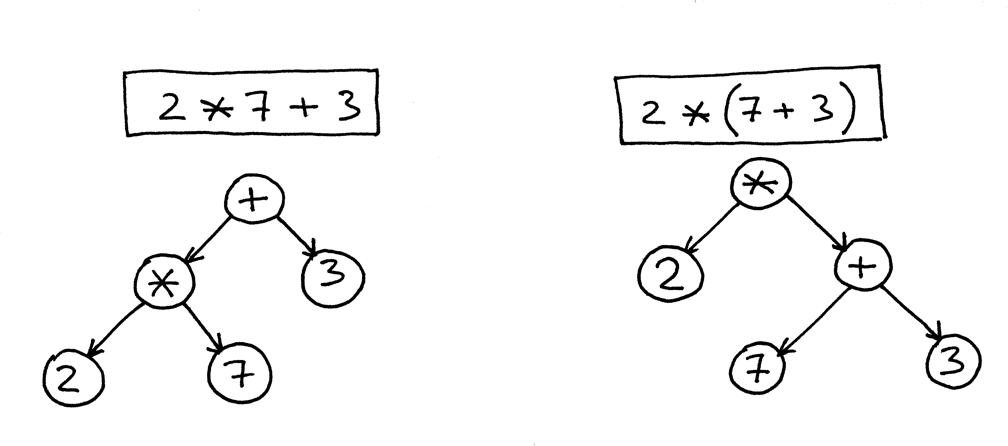
* ASTs uses operators/operations as root and interior nodes and it uses operands as their children.
* ASTs do not use interior nodes to represent a grammar rule, unlike the parse tree does.
* ASTs don’t represent every detail from the real syntax (that’s why they’re called *abstract*) - no rule nodes and no parentheses, for example.
* ASTs are dense compared to a parse tree for the same language construct.

So, what is an abstract syntax tree? An *abstract syntax tree* (*AST*) is a tree that represents the abstract syntactic structure of a language construct where each interior node and the root node represents an operator, and the children of the node represent the operands of that operator.

So far so good, but how do you encode operator precedence in an AST? In order to encode the operator precedence in AST, that is, to represent that “X happens before Y” you just need to put X lower in the tree than Y. And you’ve already seen that in the previous pictures.

Let’s take a look at some more examples.

In the picture below, on the left, you can see an AST for the expression 2 \* 7 + 3. Let’s change the precedence by putting 7 + 3 inside the parentheses. You can see, on the right, what an AST looks like for the modified expression 2 \* (7 + 3):



The AST is used intensively during semantic analysis, where the compiler checks for correct usage of the elements of the program and the language. The compiler also generates symbol tables based on the AST during semantic analysis. A complete traversal of the tree allows verification of the correctness of the program.

After verifying correctness, the AST serves as the base for code generation. The AST is often used to generate an intermediate representation (IR), sometimes called an intermediate language, for the code generation.



In our program we have 3 types of AST nodes.

**MIPS Documentation**

**Introduction:**

MIPS (Microprocessor without Interlocked Pipelined Stages) is a reduced instruction set computer (RISC) instruction set architecture (ISA).

MIPS is a RISC processor, so every instruction has the same length — 32 bits (4 bytes). These bits have different meanings according to their displacement.

**Instruction format:**

Instructions are divided into three types: **R**, **I** and **J**. Every instruction starts with a 6-bit opcode. In addition to the opcode, **R-type** instructions specify three registers, a shift amount field, and a function field; **I-type** instructions specify two registers and a 16-bit immediate value; **J-type** instructions follow the opcode with a 26-bit jump target.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Type** | **-31-                                 format (bits)                                 -0-** | | | | | |
| **R** | opcode (6) | rs (5) | rt (5) | rd (5) | shamt (5) | funct (6) |
| **I** | opcode (6) | rs (5) | rt (5) | immediate (16) | | |
| **J** | opcode (6) | address (26) | | | | |

| **Name** | **Size in bits** | **Symbol** | **Used for** |
| --- | --- | --- | --- |
| Opcode | 6 | E | Specification of instruction |
| Register specifications | 5 | s,t,d | *see below* |
| Register-immediate | 5 | R | Second part of opcode for RI and CP instructions |
| Shamt | 5 | S | Constant value for shifts |
| Immediate constant value | 16 | C | Immediate value for arithmetic and logical (AL) operations |
| Address | 26 | A | Address for jumps and procedure calls |
| Funct | 6 | f | Second part of opcode for instructions |

**Register specificators:**

Register specificators are addresses of registers. They provide numbers of registers have source data and where machine should write result of instruction. MIPS supports instructions with up to 3 registers. They are named:

* s-register *(source)*
* t-register *(target)*
* d-register *(destination)*

Additionally, MIPS RF contains a pair of special registers, HI and LO which are used to accumulate results of multiplication and division operations. In some cases they are concatenated to 2x size register notated as [HI, LO].

## **1.Add/subtract**

All R-type:

| **Name** | **Syntax** | **C code** | **Funct** | **Full format** |
| --- | --- | --- | --- | --- |
| add | add $d, $s, $t | d = s + t | 0x20 | 000000ss sssttttt ddddd--- --100000 |
| add unsigned | addu $d, $s, $t | d = s + t | 0x21 | 000000ss sssttttt ddddd--- --100001 |
| substract | sub $d, $s, $t | d = s - t | 0x22 | 000000ss sssttttt ddddd--- --100010 |
| substract unsigned | subu $d, $s, $t | d = s - t | 0x23 | 000000ss sssttttt ddddd--- --100011 |

All I-type:

| **Name** | **Syntax** | **C code** | **Opcode** | **Full format** |
| --- | --- | --- | --- | --- |
| add immediate | addi $t, $s, C | t = s + C | 0x8 | 001000ss sssttttt CCCCCCCC CCCCCCCC |
| add immediate unsigned | addiu $t, $s, C | t = s + C | 0x9 | 001001ss sssttttt CCCCCCCC CCCCCCCC |

## **2.Multiplication/division:**

All R-type:

| **Name** | **Syntax** | **C code** | **Funct** | **Full format** |
| --- | --- | --- | --- | --- |
| multiply | mult $s, $t | [HI, LO] = s \* t | 0x18 | 000000ss sssttttt -------- --011000 |
| multiply unsigned | multu $s, $t | [HI, LO] = s \* t | 0x19 | 000000ss sssttttt -------- --011001 |
| divide | div $s, $t | LO = s / t; HI = s % t | 0x1A | 000000ss sssttttt -------- --011010 |
| divide unsigned | divu $s, $t | LO = s / t; HI = s % t | 0x1B | 000000ss sssttttt -------- --011011 |
| move from HI | mfhi $d | d = HI | 0x10 | 000000-- -------- ddddd--- --010000 |
| move to HI | mthi $s | HI = s | 0x11 | 000000ss sss----- -------- --010001 |
| move from LO | mflo $d | d = LO | 0x12 | 000000-- -------- ddddd--- --010010 |
| move to LO | mtlo $s | LO = s | 0x13 | 000000ss sss----- -------- --010011 |

Special2 type (MIPS32):

| **Name** | **Syntax** | **C code** | **Funct** | **Full format** |
| --- | --- | --- | --- | --- |
| multiply and add | madd $s, $t | [HI, LO] += s \* t | 0x00 | 011100ss sssttttt -------- --000000 |
| multiply and add unsigned | maddu $s, $t | [HI, LO] += s \* t | 0x01 | 011100ss sssttttt -------- --000000 |
| multiply to low | mul $d, $s, $t | d = s \* t | 0x02 | 011100ss sssttttt ddddd--- --000010 |
| multiply and sub | msub $s, $t | [HI, LO] -= s \* t | 0x04 | 011100ss sssttttt -------- --000100 |
| multiply and sub unsigned | msubu $s, $t | [HI, LO] -= s \* t | 0x05 | 011100ss sssttttt -------- --000101 |

## **3.Shifts:**

All R-type:

| **Name** | **Syntax** | **C code** | **Funct** | **Full format** |
| --- | --- | --- | --- | --- |
| shift left logical immediate | sll $d, $t, S | d = t << S | 0x0 | 000000-- ---ttttt dddddSSS SS000000 |
| shift right logical immediate | srl $d, $t, S | d = t >> S | 0x2 | 000000-- ---ttttt dddddSSS SS000010 |
| shift right arithmetic immediate | sra $d, $t, S | d = (int32)t >> S | 0x3 | 000000-- ---ttttt dddddSSS SS000011 |
| shift left logical | sllv $d, $t, $s | d = t << s | 0x4 | 000000ss sssttttt ddddd--- --000100 |
| shift right logical | srlv $d, $t, $s | d = t >> s | 0x6 | 000000ss sssttttt ddddd--- --000110 |
| shift right arithmetic | srav $d, $t, $s | d = (int32)t >> s | 0x7 | 000000ss sssttttt ddddd--- --000111 |

## **4.Comparisons:**

All R-type:

| **Name** | **Syntax** | **C code** | **Funct** | **Full format** |
| --- | --- | --- | --- | --- |
| set on less than | slt $d, $s, $t | d = (s < t) | 0x2A | 000000ss sssttttt ddddd--- --101010 |
| set on less than unsigned | sltu $d, $s, $t | d = (s < t) | 0x2B | 000000ss sssttttt ddddd--- --101011 |

All I-type:

| **Name** | **Syntax** | **C code** | **Opcode** | **Full format** |
| --- | --- | --- | --- | --- |
| set on less than immediate | slti $t, $s, C | t = (s < C) | 0xA | 001010ss sssttttt CCCCCCCC CCCCCCCC |
| set on less than immediate unsigned | sltiu $t, $s, C | t = (s < C) | 0xB | 001011ss sssttttt CCCCCCCC CCCCCCCC |

## **5.Logical operations:**

All R-type:

| **Name** | **Syntax** | **C code** | **Funct** | **Full format** |
| --- | --- | --- | --- | --- |
| and | and $d, $t, $s | d = s & t | 0x24 | 000000ss sssttttt ddddd--- --100100 |
| or | or $d, $t, $s | d = s l t | 0x25 | 000000ss sssttttt ddddd--- --100101 |
| xor | xor $d, $t, $s | d = s ^ t | 0x26 | 000000ss sssttttt ddddd--- --100110 |
| nor | nor $d, $t, $s | d = ~ (s l t) | 0x27 | 000000ss sssttttt ddddd--- --100111 |

All I-type:

| **Name** | **Syntax** | **C code** | **Opcode** | **Full format** |
| --- | --- | --- | --- | --- |
| and with immediate | andi $s, $t, C | t = s & C | 0xC | 001100ss sssttttt CCCCCCCC CCCCCCCC |
| or with immediate | ori $s, $t, C | t = s l C | 0xD | 001101ss sssttttt CCCCCCCC CCCCCCCC |
| xor with immediate | xori $s, $t, C | t = s ^ C | 0xE | 001110ss sssttttt CCCCCCCC CCCCCCCC |
| load upper immediate | lui $t, C | t = C << 16 | 0xF | 001111-- ---ttttt CCCCCCCC CCCCCCCC |

# **Control Flow:**

## **1.Conditional branches:**

All I-type

| **Name** | **Syntax** | **PC advance** | **Opcode** | **Full format** |
| --- | --- | --- | --- | --- |
| branch on equal | beq $s, $t, C | PC += 4; if (s == t) PC += (C << 2) | 0x4 | 000100ss sssttttt CCCCCCCC CCCCCCCC |
| branch on not equal | bne $s, $t, C | PC += 4; if (s != t) PC += (C << 2) | 0x5 | 000101ss sssttttt CCCCCCCC CCCCCCCC |
| branch less than or equal than zero | blez $s, C | PC += 4; if (s <= 0) PC += (C << 2) | 0x6 | 000110ss sss----- CCCCCCCC CCCCCCCC |
| branch greater than zero | bgtz $s, C | PC += 4; if (s > 0) PC += (C << 2) | 0x7 | 000111ss sss----- CCCCCCCC CCCCCCCC |

## **2.Unconditional jumps:**

All J-type

| **Name** | **Syntax** | **C code** | **PC advance** | **Opcode** | **Full format** |
| --- | --- | --- | --- | --- | --- |
| jump | j A |  | PC = (PC & 0xf0000000) l (A << 2) | 0x2 | 000010AA AAAAAAAA AAAAAAAA AAAAAAAA |
| jump and link | jal A | ra = PC + 4 | PC = (PC & 0xf0000000) l (A << 2) | 0x3 | 000011AA AAAAAAAA AAAAAAAA AAAAAAAA |

# **Memory**

**All I-type unless noted otherwise**

## **1.Loads:**

| **Name** | **Syntax** | **C code** | **Opcode** | **Full format** |
| --- | --- | --- | --- | --- |
| load byte | lb $t, C($s) | t = \*(int8\*)(s + C) | 0x20 | 100000ss sssttttt CCCCCCCC CCCCCCCC |
| load half word | lh $t, C($s) | t = \*(int16\*)(s + C) | 0x21 | 100001ss sssttttt CCCCCCCC CCCCCCCC |
| load word left | lwl $t, C($s) | — | 0x22 | 100010ss sssttttt CCCCCCCC CCCCCCCC |
| load word | lw $t, C($s) | t = \*(int32\*)(s + C) | 0x23 | 100011ss sssttttt CCCCCCCC CCCCCCCC |
| load byte unsigned | lbu $t, C($s) | t = \*(uint8\*)(s + C) | 0x24 | 100100ss sssttttt CCCCCCCC CCCCCCCC |
| load half word unsigned | lhu $t, C($s) | t = \*(uint16\*)(s + C) | 0x25 | 100101ss sssttttt |

**Explaining the code:**

**1. Explaining Register.h:**

In register.h we define 4 functions **inrange**, **is\_reserved**, **next\_available\_register** and **free\_register**.

We define **REGISTER\_COUNT** as being 31 that will be the total number of normal registers. After that we define an enum to hold the registers adding two special registers at the end (**h1**, **lo**);

We create 5 more macros for two accumulator registers, two special registers, and the 0 register and we make a new enum for the special registers.

**2.Explaining Register.c:**

In **register.c** we define an error function that takes as parameters multiple c-style strings. Then we create two arrays one that contains our normal registers and the other one for our special registers.

The function **regstr** takes as parameter the enum declared in register.h checks if we ask for a special register after that it checks if the register exists and if all the checks work it returns the register from our register array.

The **used\_register** array marks the registers currently being used it is originally initiated with the system reserved registers and then the array changes as the registers are used by the program.

**Register\_count** returns the number of registers.

**Syscall\_req** returns the v0 register;

**Arg0\_req** returns the a0 register;

**Next\_available\_register** picks the next free register and it marks it as occupied.

**Free\_register** frees a certain register.

**Is-reserved** checks if the register is system reserved.

**In\_range** checks if the register value is between 0 and the total number of registers.

**Err** prints the error out on the screen.

**3. Explaining Codegen.h:**

Just like in **registers.h** we define an enum just that this time we used the enum to store the operation that **MIPS** can perform instead of the registers.

After that, we define an **enum called index** which is used to locate the register in an operation (first = second + third)

**OperandType** specifies the type of the operand as the name suggested it can either be a value or a register.

**YieldTyp**e specifies if we are dealing with an operand or an instruction.

After that, we define 3 structs. The first one, **Operand**, can have one of two values an int or a register and we also store the operand type in that struct. The second, **Instruction**, struct stores an instruction with 3 operands (*Ex: op1 = op2 + op3*) and the instruction type. The third struct, **Yield**, can be an operand or an instruction, it also stores the type of yield.

**4.Explaining Codegen.c:**

We define two arrays, one for registers and the other one for values, those are the values that represent the output off our program

After that, we check that the instruction served is not null then the switch statement matches the instruction type and outputs the MIPS assembly.

**Extract\_va**l and **extract\_reg** extract the value from the AST or if there is no value it outputs "missing value" to the screen.

After the value is extracted for each case the program outputs the matching assembly code.

***Final generated MIPS file***

In the end, let’s take a small example of a part of a code, see what is generated and talk about how to read the generated file.

.text

.globl main

exit:

li $v0, 10

syscall

main:

# VAR

li $a2, 3

# VAR

li $a3, 5

# VAR

# =

# +

add $t0, $a2, $a3

# VAR

li $t1, 0

# VAR

li $t2, 0

# WHILE

L001:

# <

slti $v1, $t2, 5

li $a1, 0

beq $v1, $a1, L002

# =

# +

addi $t1, $t1, 1

# =

# +

addi $t2, $t2, 1

j L001

L002:

j exit

**Generated file**

var a=3;

var b=5;

var s;

s=a+b;

var c=0;

for ( var i = 0 ; i < 5 ; i = i + 1 )

{

c=c+1;

}

**Code example**

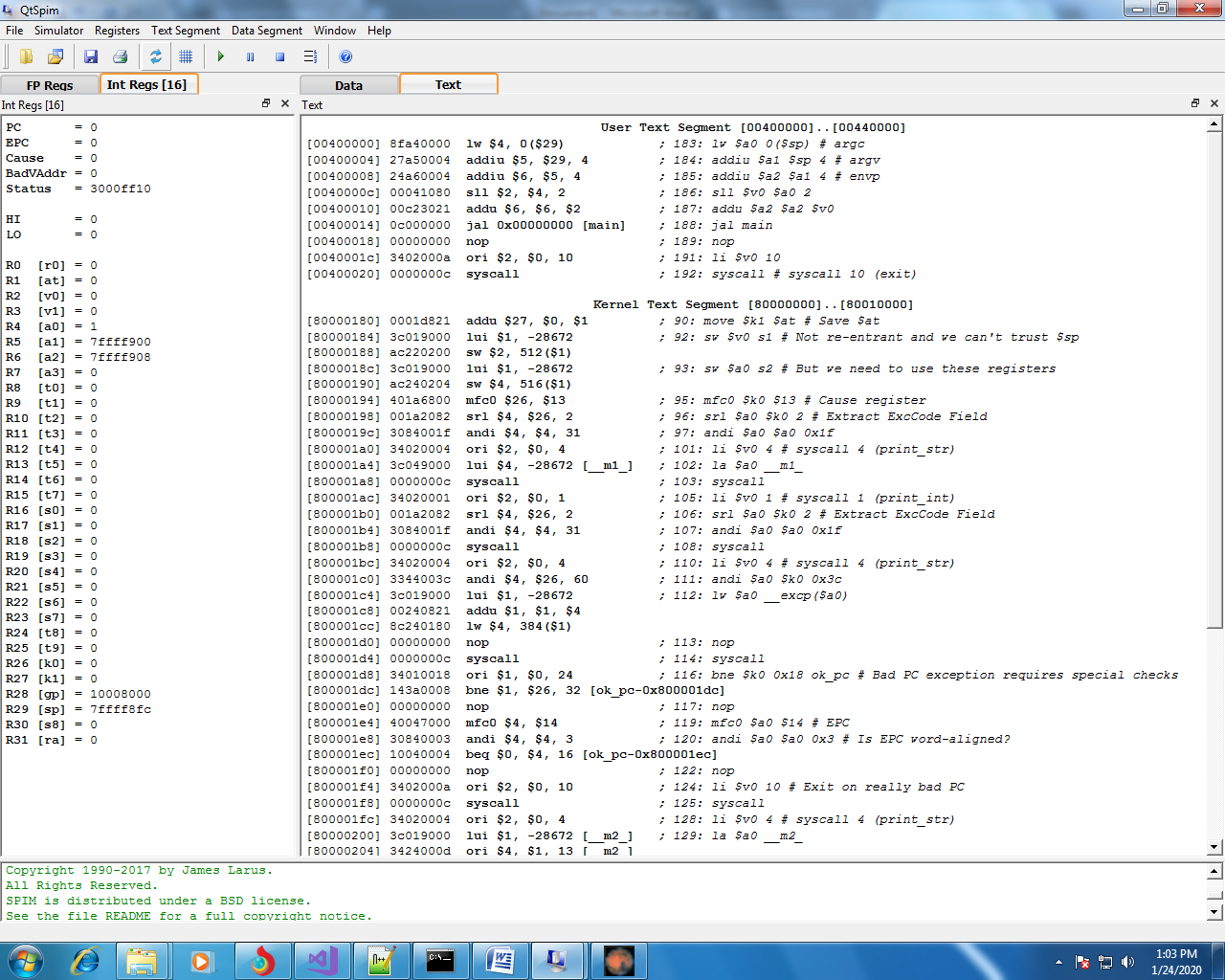
***Tools for testing output of the compiler***

I present two tools for testing the output of the compiler:

1. QtSpim
2. MARS
3. ***QtSpim***

QtSpim is a MIPS R3000 processor simulator.

This is the main window of the simulator:



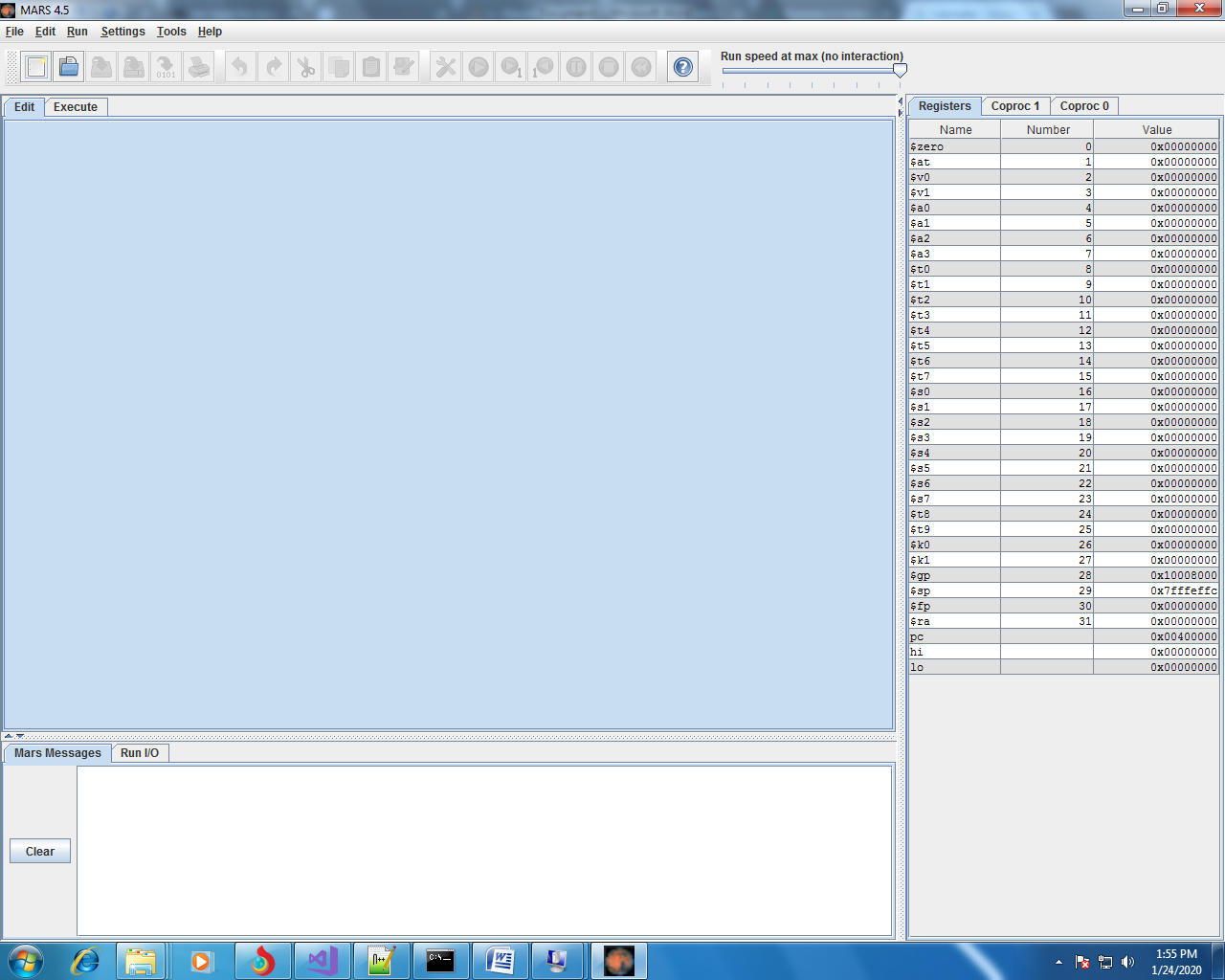
To test a file you open the file menu and select from it “Reinitialize and load a file” and a popup window appear where you can select your file. If the file contain any error a popup dialog appear that tell you first error occurred. To run your code instruction by instruction from menu “simulator” select “Single step” or press F10.

For this simulator to work with a file, it must have a main function defined in that file.

***2. MARS***

Mars is an outdated mips simulator / editor that reads instruction after instruction the code and does not require a main function to work. This means you can test pieces of code without calling them from the main function.

This is the main window of MARS:



To test a file you open the file menu and select from it “Open…” and a popup window appear where you can select your file. To run a file from menu “Run” select “Assemble” and if no error occurred the Execute tab will open and the “Run the current program” and “Run one step at time” will be avabile.

1. a basic lexical unit of a language consisting of one word or several words, the elements of which do not separately convey the meaning of the whole. [↑](#footnote-ref-1)