Formal Languages and Compilers

github.com/martinopiaggi/polimi-notes

2022-2023

Contents

1 Regular Expressions

The family of **regular languages** *(also called or type* 3*)* is the simplest language family.

- Regular expressions can be ambiguous if there are multiple structurally different derivations that result in the same sentence. Sufficient conditions for ambiguity exist.
- Regular expressions can be extended with operators like union, concatenation, and star.
- A language on an alphabet is regular if it can be defined by a regular expression. So only if it is defined by concatenation, union and star over the elementary languages of L ($\{a_1\}, \{a_2\}, \{a_3\} \ldots$).
- Regular expressions have limits, as they cannot represent certain languages, such as those with unbalanced nesting or varying numbers of elements. To represent these languages, **generative grammars** must be used instead.

1.1 Regex

Regex basics: - `x` the `x` character - `.` any character except newline - `[xyz]` means `x` or `y` or `z` - `[a-z]` any character between `a` and `z` - `[^a-z]` any character except those between `a` and `z`

Said `R` a regular expression: - `RS` concatenation of `R` and `S` - `R|S` either `R` or `S` - `R*` zero or more occurrences of `R` - `R+` one or more occurrences of `R` - `R?` zero or one occurrence of `R` - `R{m,n}` a number or `R` occurrences ranging from `n` to `m` - `R{n,}` at least n occurrences - `R{n}` exactly n occurrences of `R`

2 Free grammars

Regular expressions can provide lists but not nesting (recursion). A generative grammar is a set of simple rules that can be repeatedly applied to generate valid strings. The grammar defines languages through rule rewriting and the repeated application of these rules. A context-free grammar, also simply known as a free grammar or a type 2 or BNF (Backus Normal Form) grammar is defined as:

$$G =< V, \Sigma, P, S >$$

where:

- V is the set of non terminals symbols
- Σ is the set of terminals symbols
- P is the set of productions (sometimes are indicated as R (rules))
- S is the axiom, a particular symbol of V

Note that the grammar is said to be "free" when the production rules apply equally to all occurrences of a nonterminal symbol in the grammar, without regard to the **context** in which it appears (indeed we can call them context-free). This distinguishes it from a context-sensitive grammar, where the production rules depend on the context in which the nonterminal appears.

- Recursion is the key to create infinite grammars. The necessary and sufficient condition for a grammar to be infinite is that there is a recursive derivation. Note that a grammar have a recursive derivations iff the corresponding graph has a circuit.
- A rule can be left recursive or right recursive or both
- The concept of left recursive or right recursive is quite important in order to generate a left-associative operator for example. For a left-associative operator is necessary to use left-recursive rules.

```
A -> A terminal | X
```

- Clean grammars are grammars without a circular axioms.

Classificazione regole importanti

Given a rule $A \to \alpha$ with $A \in V$ (V are non-terminal symbols set) and $\alpha \in (V \cup \Sigma)^*$ (where Σ is the set of terminal symbols of the grammar) we can classify the production itself according to many definitions. **Some** of the most important are:

- Recursive rule on the left: the production is called a recursive rule on the left if its left part appears as a prefix of the right part $A \to A\delta$
- Recursive rule on the right The production is called a recursive rule on the right if its left part appears as a suffix of the right part $A \to \delta A$

There are many normal form; one of the most important is indispensable for the top-down parsers to be studied later in the course. This normal form is termed not left-recursive and it is characterized by the absence of left-recursive rules or derivations.

2.0.1 Syntax tree

Given a grammar G and a string in $L(G)$ we can build a syntax tree of the string where the root is the axiom S, the leafs are the sequence of the symbols of the string and all the branches are the productions used to generate the string. Two possible algorithms to build syntax trees:

- ELL: top-bottom
- ELR: bottom-up

2.1 Ambiguity

With the definition of Syntax Tree we can also specify what is an ambiguity in the context of grammars. A sentence is ambiguous if it admits different syntax trees. The number of distinct trees is also the degree of the ambiguity.

> Avoid the ambiguity in the grammar design phase.

2.2 Grammar classification

Chomsky proposed a classification of grammars, thus establishing an order based on their generality and defining the "types" from 0 to 3, where type 3 represents the least general grammars and type 0 is the most general case. In this course, we will only be dealing with type 2 and 3 grammars. Each type of grammar is a subset of the one which precedes it.

2.2.1 Type 0 grammars

- Family of recursively enumerable languages
- Associated with Turing machines and generates recursively enumerable languages.
- The languages generated are not generally decidable.
- Rules are of the most general type.

2.2.2 Type 1 grammars:

- Family of context-sensitive languages
- Associated with Turing machines with tape of length equal to that of the string to be recognized.
- Generates context-sensitive languages, which are always decidable.

2.2.3 Type 2 grammars

- Family of unrestricted languages
- Associated with non-deterministic stack automata.

- Generates unrestricted languages, which are always decidable and which we have already extensively analyzed.

2.2.4 Type 3 grammars

- Family of regular languages
- Associated with finite-state automata.
- Generates regular languages.
- We can classify them to unilinear right or left

Grammar	Rule type	Language Family	Recognizer Model
Type 2	$A \to \alpha$ with A non-terminal and α can be anything (non-terminal/terminal)	context-free lang / BNF lang	Pushdown Automaton (non-deterministic)
Type 3	$A \to uB$ or $A \to Bu$ (not both) with A non-terminal and B nonterm or ϵ	Regular or rational lang	Finite state automaton

2.2.5 EBNF

Extended Backus Normal Form (EBNF) is just a variant grammar that contains exactly one rule (i.e. one rule for each nonterminal); each rule has a different nonterminal on the left side and has a regular expression of alphabet on the right side, in which derived operators such as cross, power and optionality can also appear.

```
EXPR -> IMP [imp IMP]
IMP -> TRM (or TRM )*
TRM -> FCT (and FCT)*
FCT -> [not] EQU
EQU -> OBJ ['=' OBJ]
OBJ -> "x" | "y" | "z" | "(" EXPR ")"
```

2.2.6 Dyck Language

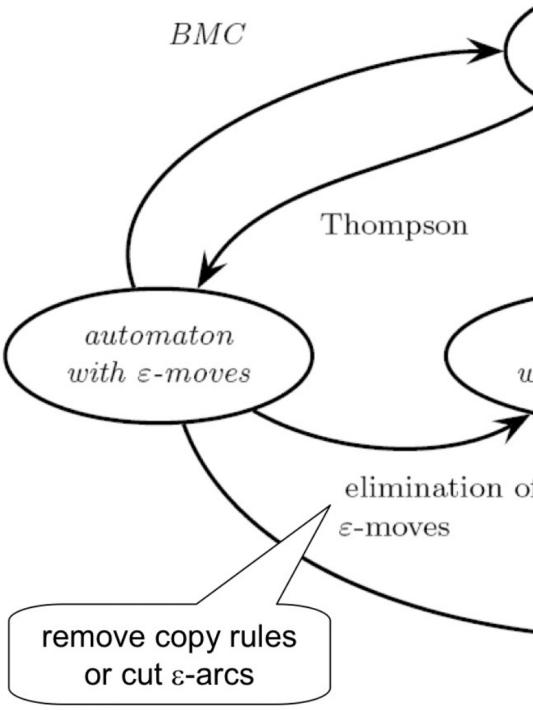
Model of languages well-parenthesized:

$$\Sigma = \{a, a', b, b'\}$$
$$S \to aSa'S \,|bSb'S|\, \varepsilon$$

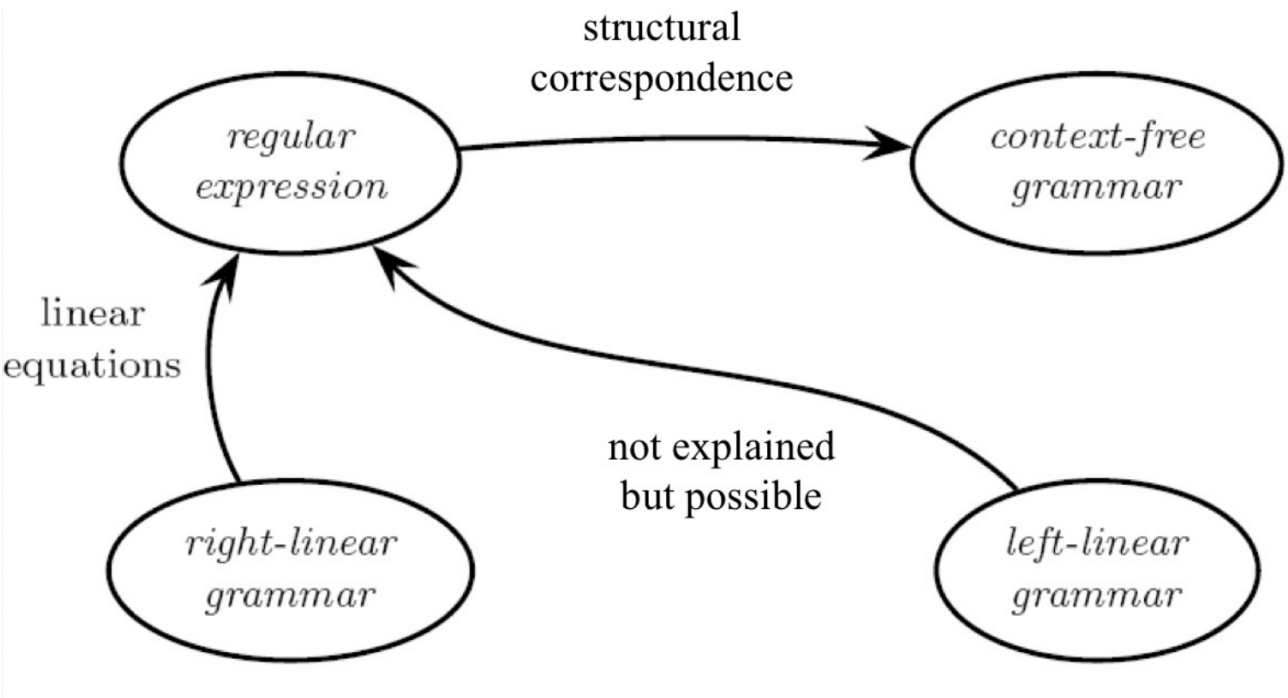
Dyck language is free but not regular.

2.3 Linear language equations

Linear languages can be represented as a set of linear equations. Every rule corresponds to a linear equation.



The system of equations can be solved using substitution and the Arden Identity.



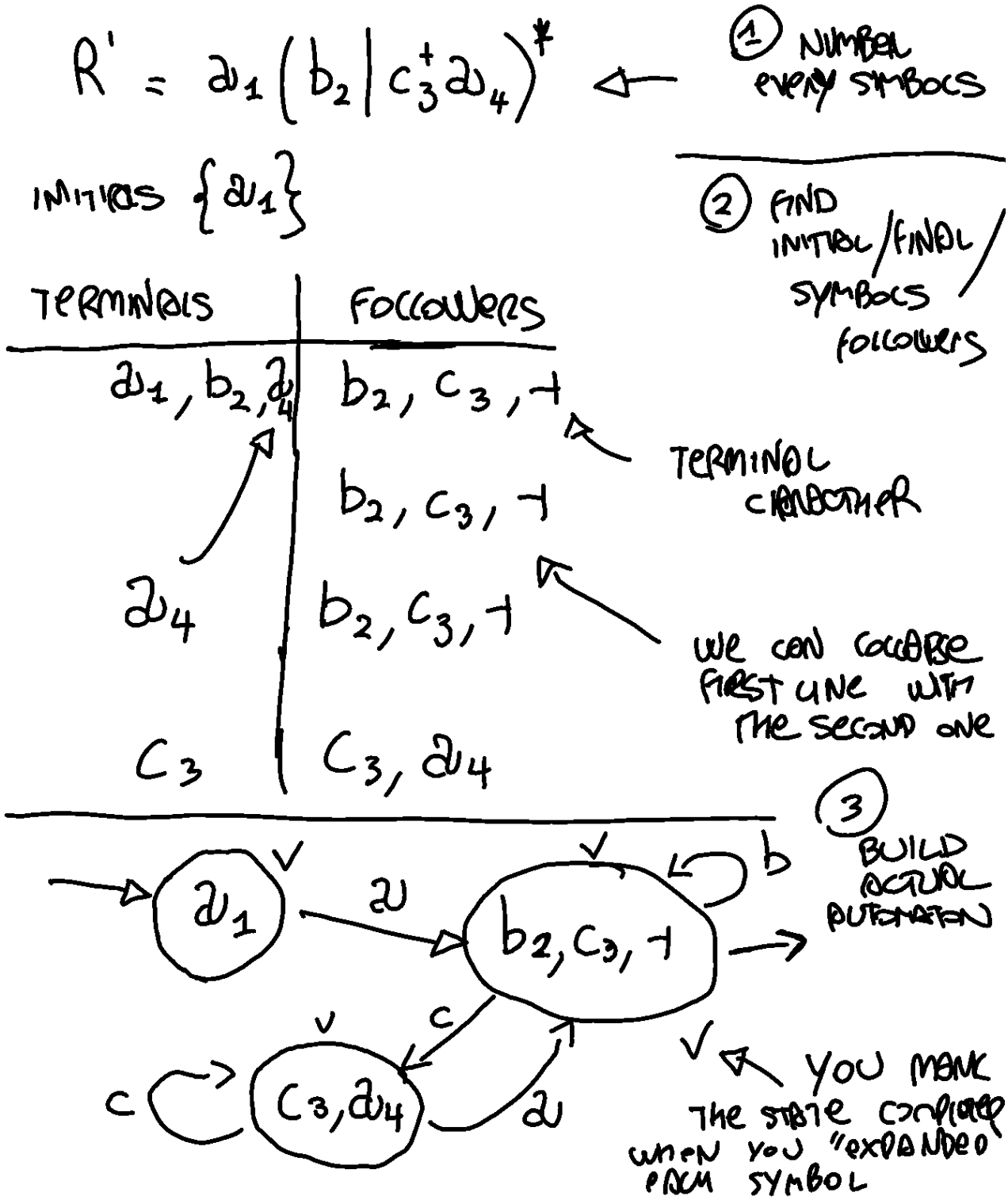
3 Berry Sethy

Before starting:

- initials: symbols at the start of the string
- terminals: not the symbols that appear at the end of a string! But all the symbols that are not initials.
- followers: all the symbols that can follow a terminal

Convert a regular expression to finite state automaton.

1) rename all the symbols giving a number to all of then
2) find initial / terminals / followers symbols
3) eventually collapse lines in the table
4) build actual automaton and in each state you put in the followers



Procedure to check if it s minimal is to use a matrix where each column is a state and each line a transition letter.

Berry Sethi can work also viceversa following the exact same rules.

4 BMC algorithm

The BMC-algorithm (Brzozowski Mc-Kluskey) is used to convert a finite automata into a regular expression. It's very simple: eliminate all internal states once at time while adding compensatory moves labeled by regular expressions. The order in which internal states are deleted is irrelevant.

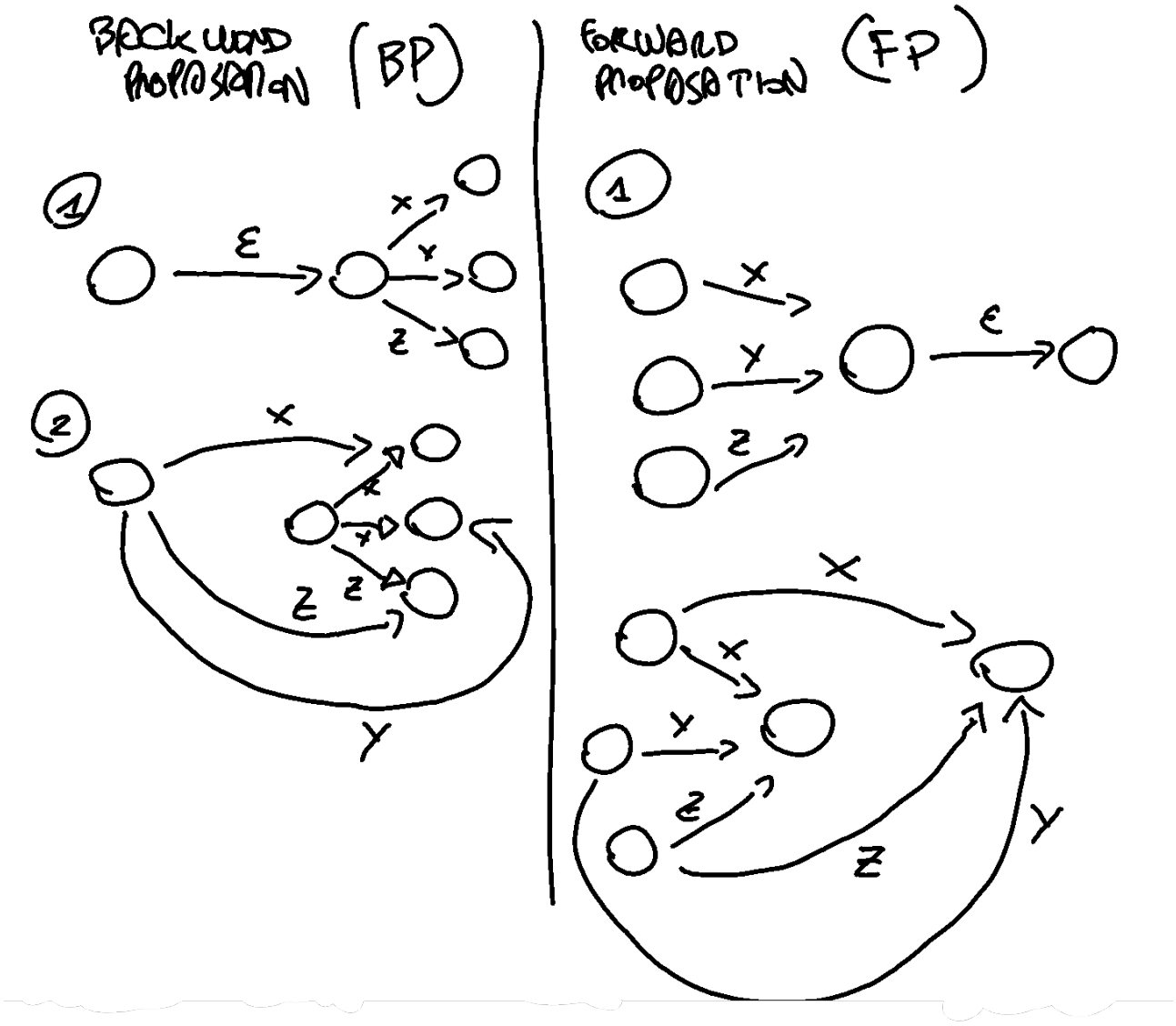
It is used to convert an automata into a regular expression.

1) you add an initial and final state
2) then you iteratively:
    1) choose a random node X
    2) replace the arrows **concatenating the symbols** on the arcs with this rule: create an arc from A to B if there are an arc from A to X and an arc from X to B.
    3) repeat until the two artificial nodes are linked
3) the reg expression will be on the only final arc. If there are multiple final arcs just use | operator.

4.0.1 Two possible way to eliminate ϵ transition :

- backword propagation (BP)
- forward propagation (FP)

note that eliminating the ϵ - moves produce indeterministic automatons.



we cut transition 2 $\epsilon \rightarrow$ 3 and we back-propagate the outgoing arc

Humans can tolerate ambiguity, computers and compilers no.

4.1 Explained to a child

And this is what actually happen in reality and in our exercises. If a grammar which is not LR is parsed by a LR tool this two things can happen in practice:

- reduce reduce conflicts: basically the compiler gets confused because it sees two possible ways to group things together
- shift reduce conflicts: the compiler this time gets confused because it doesn't know if it should take the next block and build with it right away, or if it should wait and try to group it with another block later. The typical example of this is the "dangling else problem" (it's an ambuiguity problem).

Exists also LL, which is another way to parse the input. LL parsers use a top-down parsing approach and have the property of not allowing left-recursion in their grammar. Left-recursion would cause the parser to enter into

an infinite loop, as it would continually expand the same non-terminal symbol without making progress. LL parsers are also less expressive than LR parsers.

The main goal of a syntax analyzer or parser is to determine if a source string is in the language $L(G)$ of a given grammar G, and if so, to compute a derivation or syntax tree. We know the same (syntax) tree corresponds to many derivations, notably the leftmost and rightmost ones, as well as to less relevant others that proceed in a zigzag order.
Two important parser classes:

- Top-down analysis (LL(k) or predictive): Constructs leftmost derivation by starting from root and growing tree towards leaves. Each step is a derivation.
- Bottom-up analysis (LR(k) or shift–reduce) : Constructs rightmost derivation in reverse order, from leaves to root. Each step is a reduction.

If the string is not in the language, the parser stops and prints an error message.

4.2 Bottom-Up Deterministic Analysis ELR(1)

We start to consider EBNF grammars and ELR(1) (The acronym means Extended, Left to right, Leftmost, with length of look-ahead equal to one). ELR(1) parsers implement deterministic automaton with pushdown stack and internal states (called macro-states).

- Candidate is a way that ELR uses to verify the match is correct. A pair `(state, token)` is named a candidate (also known as item).
- Macro-states consist of a set of candidates.
- A closure is the set of all candidates that can be find starting from the initial state and ''looking-ahead". To calculate the candidates for a given grammar or machine net, we use a function traditionally named closure. which says that the end-of-text character is expected when the entire input is reduced to the axiom
- A state can have more candidates. ILR looks multiple candidates in parallel. $C =< q, r >$ where q is a state . A closure is all possible candidates of a given states.
- Automaton performs two types of moves: shift and reduce.
  - Shift move: Reads incoming character, computes next macro-state, pushes token and next macro-state onto stack.
  - Reduce move: Occurs when top-most stack symbols match recognizing path in machine and current token is in current look-ahead set. At this point, it expands the syntax tree, updates stack by popping matched top-most part, pushing recognized nonterminal symbol and next macro-state.

In the exercises we will check the determinism of all this stuff checking conditions related to the shift and reduction movements (conflicts). Doing this means to check if the grammar can be recognized.

4.3 Conflicts and condition for ELR(1)

There are three problems that must be verified for determinism in the pilot graph:

- Shift-reduce conflict: transition to another state with a character x and there is also a final state with lookahead x . "the pilot can't know if it has finished or not".
- Reduce-reduce conflict: in every configuration, at most one reduction is possible so we have a conflict when a state has two or more moves with the same lookahead.
- Convergence conflict: multiple transitions reach same state.

If the determinism test is passed, the PDA can analyze the string deterministically.

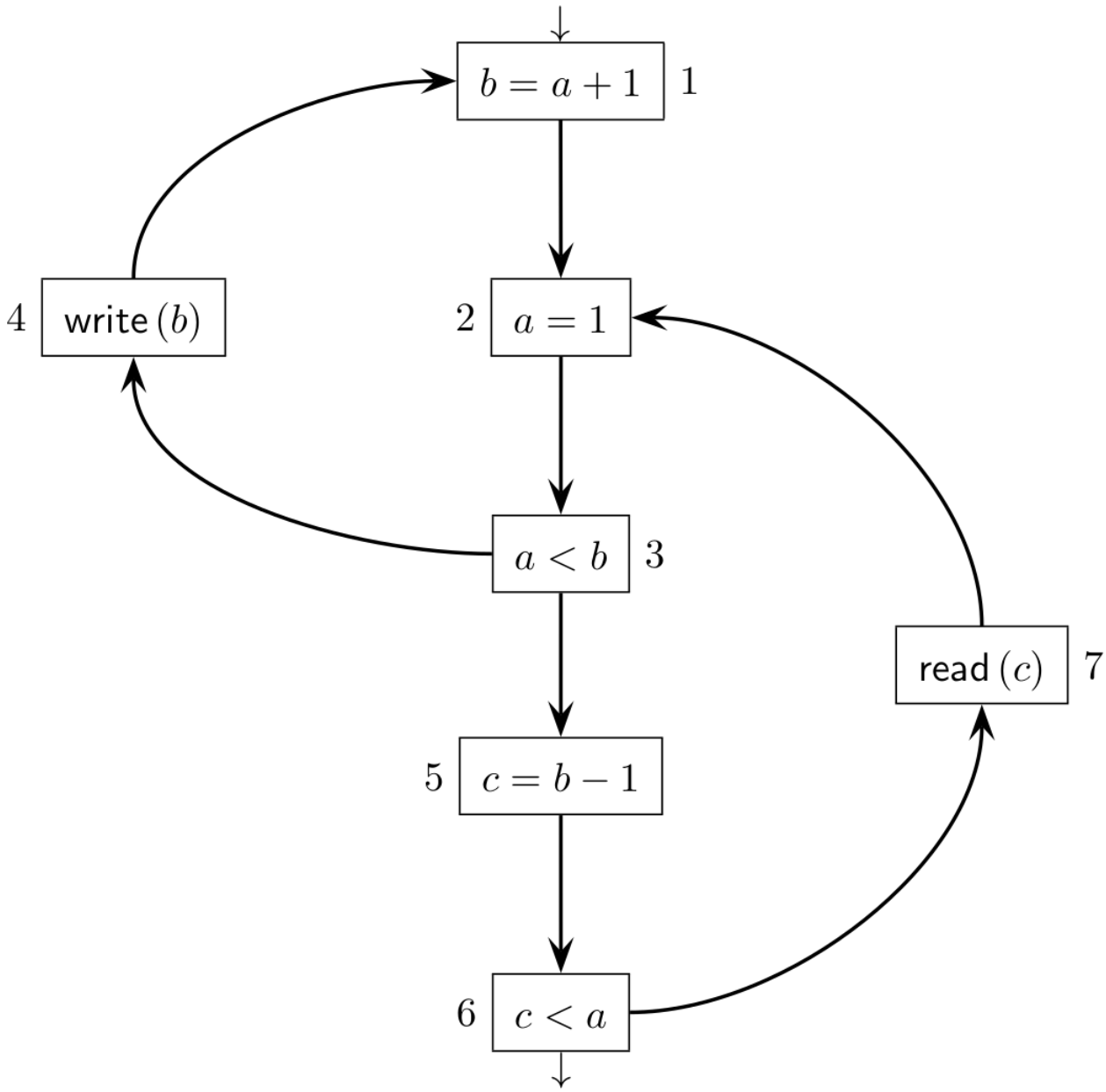
4.4 ELL

$ELL(1)$ is a simple and flexible top-down parsing method for $ELR(1)$ grammars with additional conditions. A machine net M (so the grammar) meets the $ELL(1)$ condition if the following three clauses are satisfied:

1) there are no left-recursive derivations
2) the net meets the $ELR(1)$ condition, i.e., it does not have either shift–reduce, or reduce–reduce, or convergence conflicts
3) the net has the single-transition property (STP)

4.4.1 Using the guide-sets

A descending parser is said to be $ELL(1)$ if **at every point** in the machine network where there are multiple possible paths to take, **it is possible to determine which path to choose by looking at the next character in the input**. This means that the guide sets (sets of characters that can appear next) are disjoint for the different paths at the junction. So it's possible to manually check the guide sets at the exam without the pilot graph and using the machine nets looking branching point on each state and checking that the look-ahead sets of the outgoing transitions are disjoint.



In other words, a variable is live out of a certain node if some instruction that may be successively executed, makes use of the value the variable has in the former node.

Static analysis

Static analysis of programs is a set of operations that are performed a priori, or not during the execution of the program itself, which aims to extract information from the source code or an intermediate form of code, which then allows to perform some advantageous operations.

In addition to recognition and parsing, many language processing tasks involve some form of transformation on the original sentence. For example, a compiler translates a program written in a high-level programming language such as Java, into the machine code of a specific microprocessor.

- Consider only the simplest instructions, such as those involving a scalar variable and constant.
- Focus on assignment to a variable and simple unary arithmetic, logic, or relational operations.
- Only consider intraprocedural analysis (not interprocedural).
- Every graph node represents a program instruction (statement).
- If instruction p is immediately followed by q at execution time, the graph has an arc directed from p to q, with p being the predecessor of q.
- The first instruction of the program is the graph input node (initial node).
- An instruction without successors is a graph output node (final node).
- Unconditional instructions have one successor, whereas conditional instructions have two or more successors.
- An instruction with two or more predecessors is a confluence node (also called merge or join node).
- Keep in mind that the control graph is not a complete and faithful representation of the program.
- The logical value of a condition (true or false) that selects the appropriate successor of a conditional instruction is not represented.
- Replace the assignment operation with abstractions such as:
  - A variable defines that variable.
  - Referencing a variable in the right member of an assignment, an expression, or a write operation uses that variable.

Liveness of a variable

A variable is alive in some point p of the program if some instruction reachable from p will use the value of the variable.

Each define of any variable it's like a reset/breakpoint in the liveness of the variables.

Liveness information is an hint for the compiler to "is this variable necessary to the program? Can I remove it?" . So basically liveness is used to optimize memory allocation.

At the end you take the "in" and "out" sets and write them near the blocks.

You make analysis for example on reaching definitions: For each variable we check where it's been defined.

A block A dominates a block B if to reach block B it's necessary to pass from block A .

Where p is an arbitrary block.

$$\text{in}(p) = \text{use}(p) \cup (\text{out}(p) \setminus \text{def}(p))$$

$$\text{out}(p) = U\text{in}\,(p_{\text{successors}}\,)$$

live out of each set must be coherent with the in set of any p successors.

The algorithm consists basically to apply the formulas through many iterations until we reach a "fixed point": the in and out set doesn´t change anymore from a set to another one.

5 Translation semantics

A translation can be thought of as a function or, more generally, a mapping between the strings of the source language and the strings of the target language. Like string recognition, there are two main approaches to translation. The generative approach involves creating a mapping between the source and target languages, while the other approach uses a transducer, which is similar to a recognition automaton, but with the added capability to emit a target string.

Syntactic translation, or compilation, consists of obtaining a new string, called an image, from a certain source string. The two strings can belong to different languages, called source language and image language. The machine that does this is called a syntactic translator or compiler. The translation can be ambiguous or

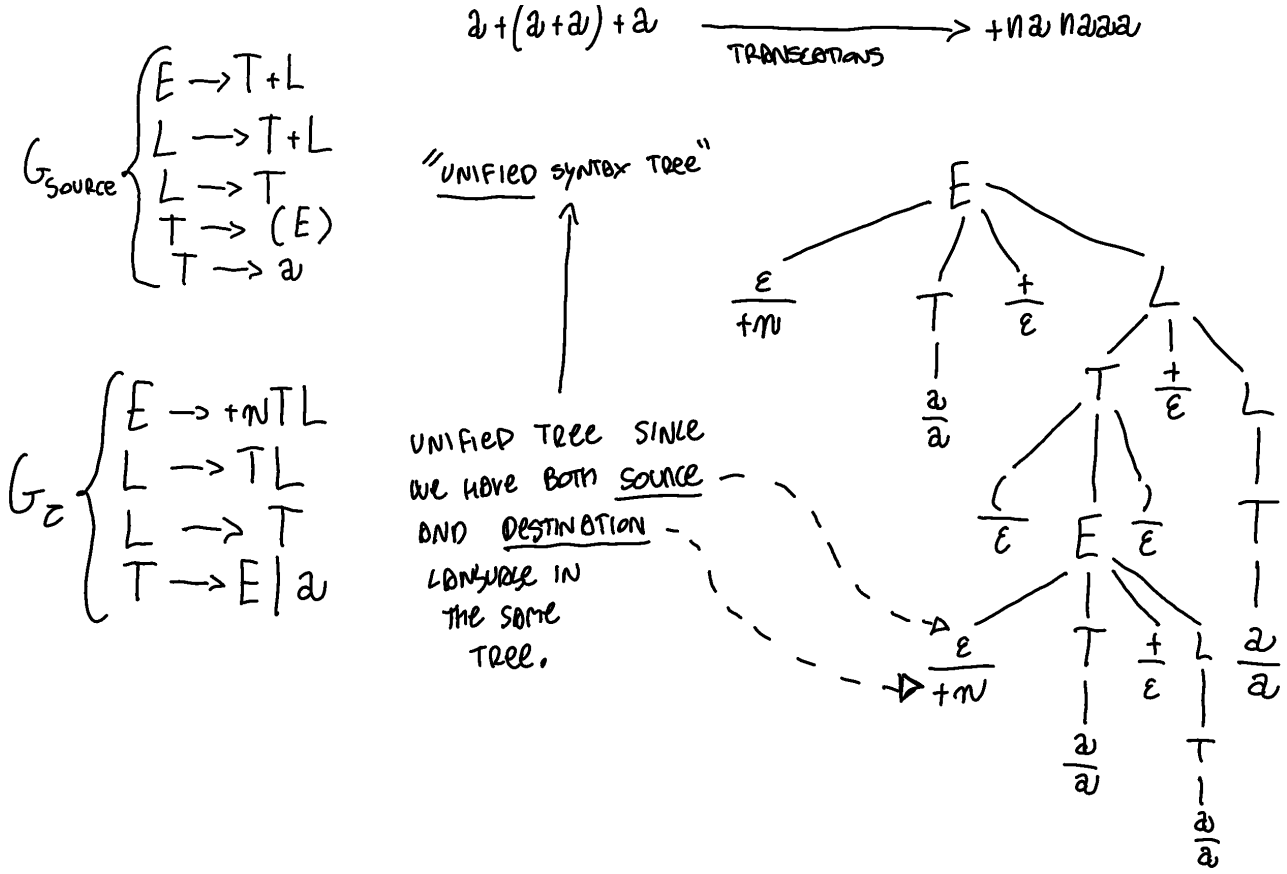
multi-valued if the same source string has multiple images in target. Inverse translation is simply the inverse relation of translation. The properties of a translation can be injectivity, surjectivity and biunivocity. There are two types of translation: purely syntactic and syntax-driven. Purely syntactic translation relies solely on source language syntax, while syntax-driven translation also uses some variables. Purely syntactic translation can be formalized using a translation grammar, a 2I-automaton, an IO-automaton or a regular translation expression.

5.1 Translation grammars

Translation grammars Syntactic schemes for purely syntactic translation A purely syntactic translation can be formalized in various equivalent ways:

1. Through a translation grammar
2. Through a 2I-automaton (also called a Rabin & Scott machine).
3. Through an IO-automaton.
4. Through a translation regular expression. Let's start here by looking at the first method. Translation grammars Given the source language and the target language, and given two grammars that generate these languages, if the following conditions are met:
5. There is a one-to-one correspondence between the rules of and ;
6. The corresponding rules of the two grammars differ only by the terminals that appear in them;
7. The corresponding rules of and contain the same non-terminals, in the same order. Then a translation grammar ( ) can be constructed, which is actually just a more concise representation of the given pair of grammars. The translation grammar simply involves rewriting the same rules of and , by replacing the terminals with symbols of the type $\frac{a_1}{a_2}$.

Example:



6 Attribute grammars

Attribute grammars are basically the theory behind Bison

A "grammar with attributes" is a grammar in which one or more procedures can be associated with each rule to calculate the attributes associated with the nonterminals that appear in the rule itself. Attributes are associated with various nonterminal or terminal symbols that appear in the grammar. The tree with the addition of these attributes is called a decorated tree.

Attributes left and right

- Left or synthesized attributes: an attribute is said to be left or synthesized if it allows calculating an attribute associated with the non-terminal present as the left part of the rule; it will therefore be of the type:
- Right or inherited attributes An attribute is said to be right or inherited if it allows calculating an attribute associated with one of the non-terminals in the right part of the rule; it will therefore be of the type:
- An attribute cannot be both left and right at the same time.

Synthesized attributes are computed from the attributes of the children of a nonterminal.

Inherited attributes are computed from the attributes of the parent nonterminal and the attributes of the children.

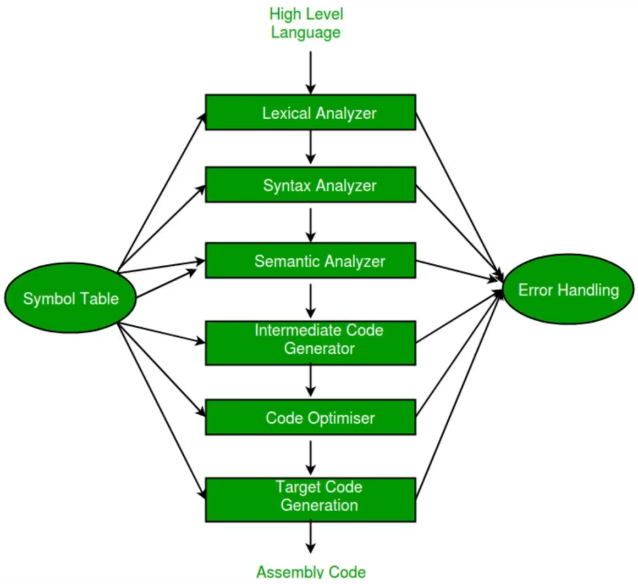
We generally use a table to indicate an attribute grammar. We use numbers to distinguish the symbols in the "given grammar".

The important stuff to look in the attribute grammars is to look eventually circular dependency. Indeed if an attribute grammar is acyclic , it's also correct.

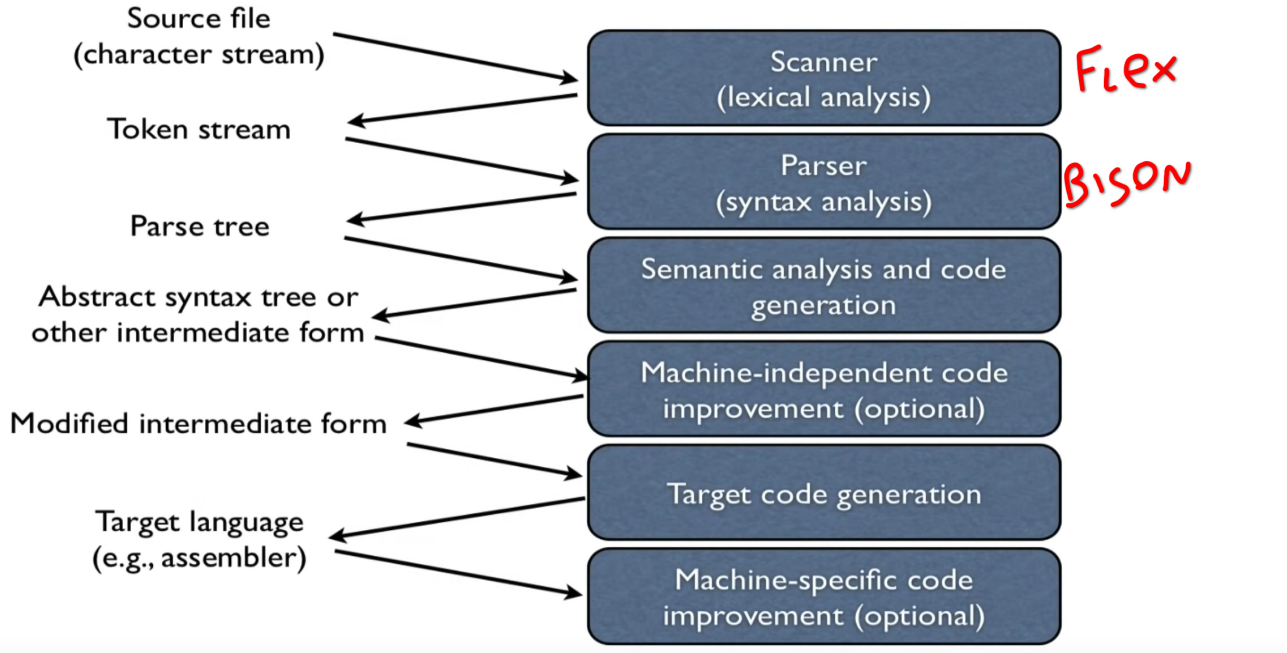
Additionally, it is of type one-sweep, meaning that the attribute values can be computed in a single pass over the parse tree, with each attribute value depending only on the attribute values of its parent, children, and itself. This makes the attribute grammar easy to evaluate and understand. A dependency graph can be used to visualize the attribute dependencies and further confirm that the attribute grammar is of type one-sweep.

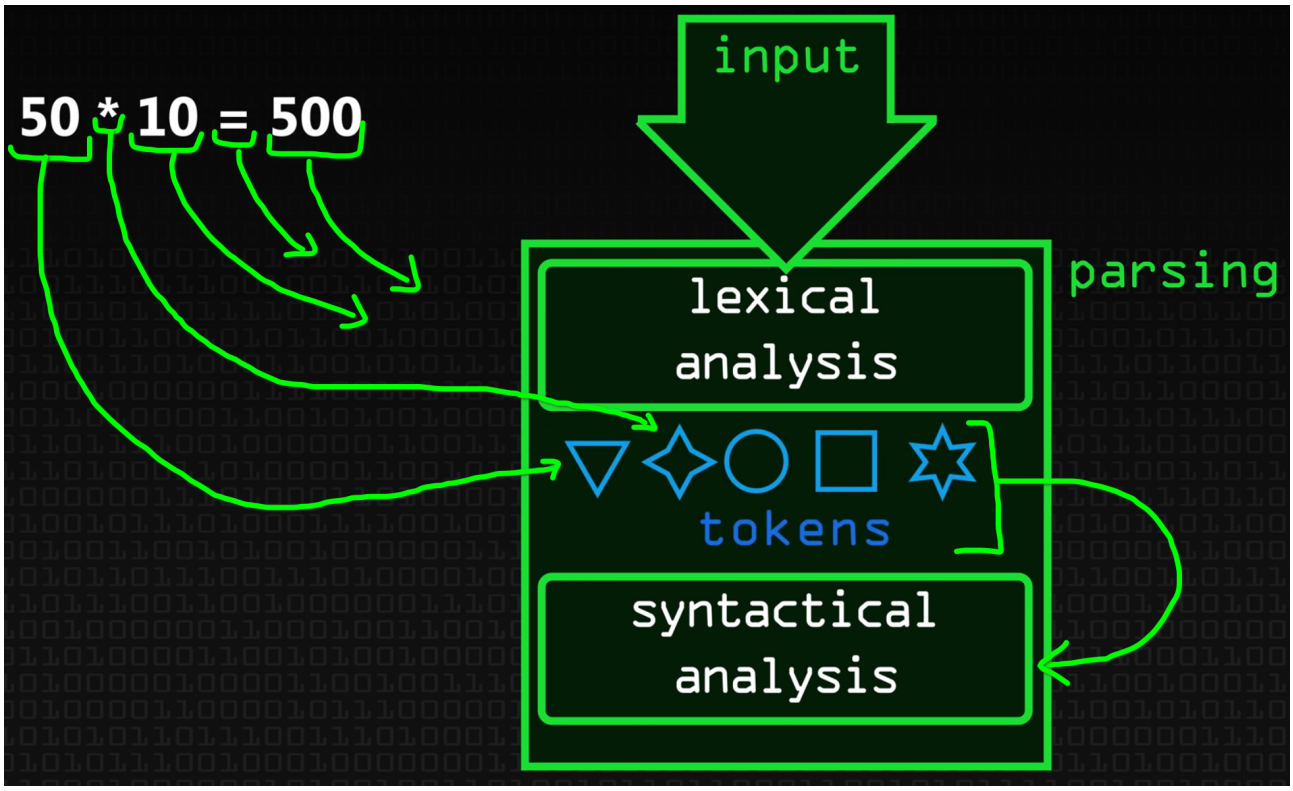
2 sweeps grammar ???

7 Lab ACSE



There are several variants of LR parsers, one of them are **LALR** parsers. LR (and LALR) parsers can be generated by a parser generator from a formal grammar defining the syntax of the language to be parsed. This is what Bison (GNU Parser Generator) does: it's not a parser but a parser generator. Bison uses the LALR(1) parsing algorithm, LALR(1) stands for "Look-Ahead LR(1)" which is an efficient bottom-up parsing algorithm. LALR(1) builds a parse tree for a given input string by starting at the leaves and working its way up to the root. It's an efficient algorithm for parsing context-free grammars. LALR(1) languages are strictly less expressive than general context-free languages, but are more efficient to parse.



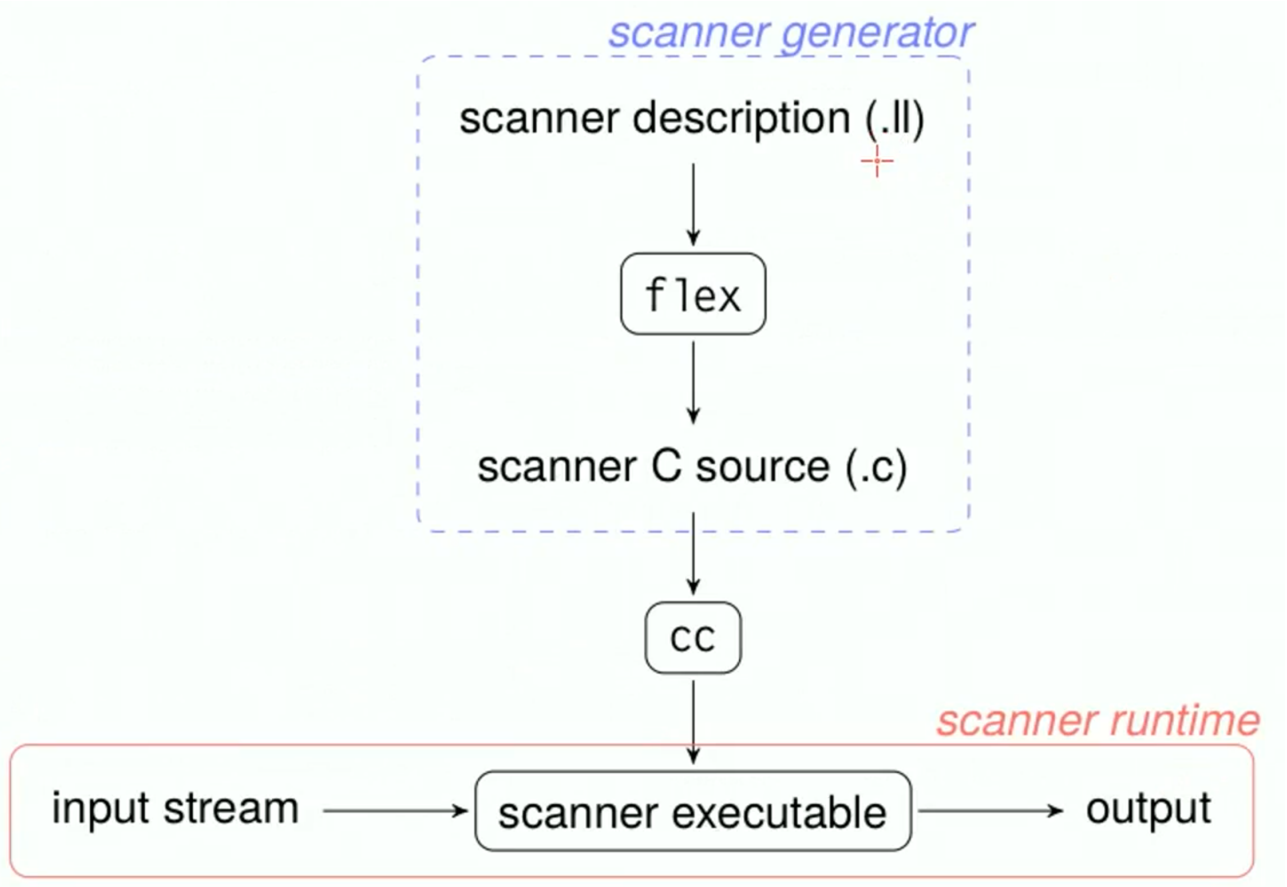


Mindset: in this kind of exercises we are not actually doing assignments, we are just writing the instructions that will produce the assignment when the program will be executed. We are translating the program, not executing. Compilation is a form of planning. No space for its value is reserved in the memory of the compiler, however, the compiler knows in which register or memory location the variable will be at runtime.

```
sudo apt update && sudo apt install -y build-essential flex bison
```

In acse the assembly code is generated while the parser (generated by Bison) is parsing (using the Bison semantic rules).

7.1 Flex



It's implemented as non-deterministic finite state automaton.

- definitions: where you can declare useful REs
- rules: most important part where you bind RE combinations to actions
- user code: C code

Each part is separated by **%%**

Longest matching rule The flex scanner in case of more than one match (with more one rules) the longer match will always win: if a longer regular expression match an expression and then others regular expression match subsets of the expression, the longer rule willl have precedence.

the first rule

Flex breaks up the input into individual tokens that are then passed to the parser (in our case we use Bison) for further processing.

Flex -> we do lexical analysis: from words to tokens In particular Flex generates a scanner (which recognizes tokens in the stream of characters and maybe decorate them in order to provide additional info).

A flex file is structured in this way (three sections separated by %%).

```
Definitions
//declare useful REs
%%
Rules
```

```
//bind RE combinations to actions
%%
User code
//C code (generally helper functions)
```

Each rule basically corresponds to a token to be recognized: in particular it links a token with an action to perform when the token is matched.

8 Bison

It reads a specification of a grammar for a particular programming language or input format, and generates a parser which can recognize and parse input written in that language or format.

- prologue: useful place where to put header file inclusions and variable declarations
- definitions
- rules
- user code

Bison file structure is similar to Acse.

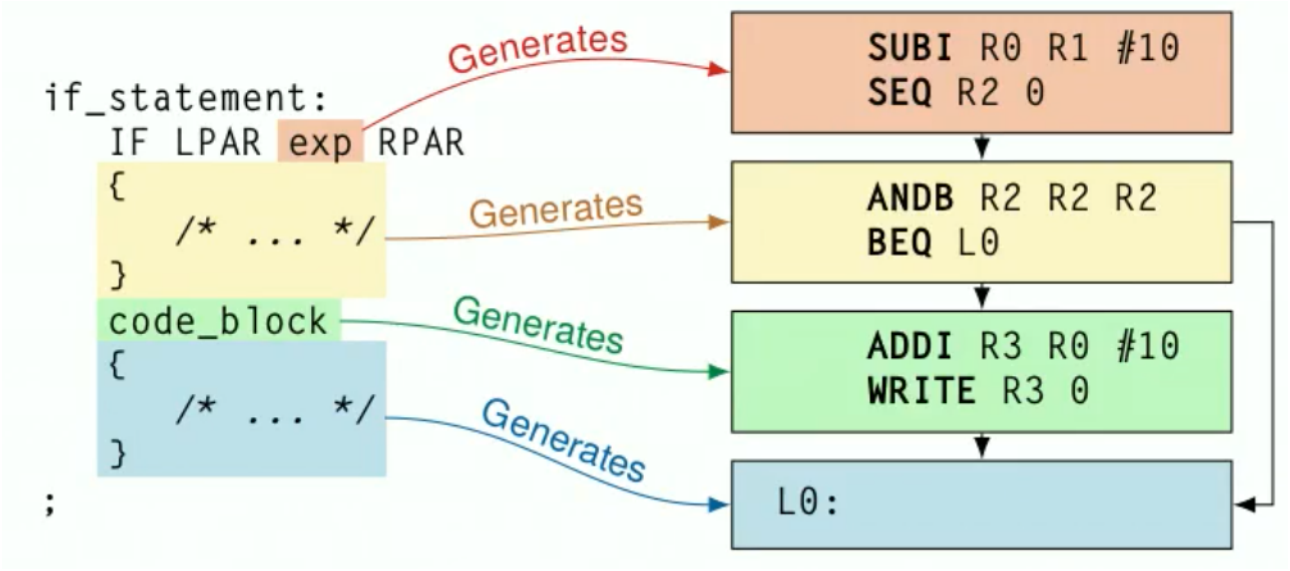
```
%{
//prologue and headers
}%
// definitions
%%
//rules like a context-free grammar
not_terminal : terminal1
                    | terminal2 TOKEN_A
                    | terminal3 { /* C code */ }
%%
//user code
```

Very similar to grammar ... uppercase tokens are non terminals and lowercase tokens are terminals symbols.

You can add semantic actions for each grammar rule.

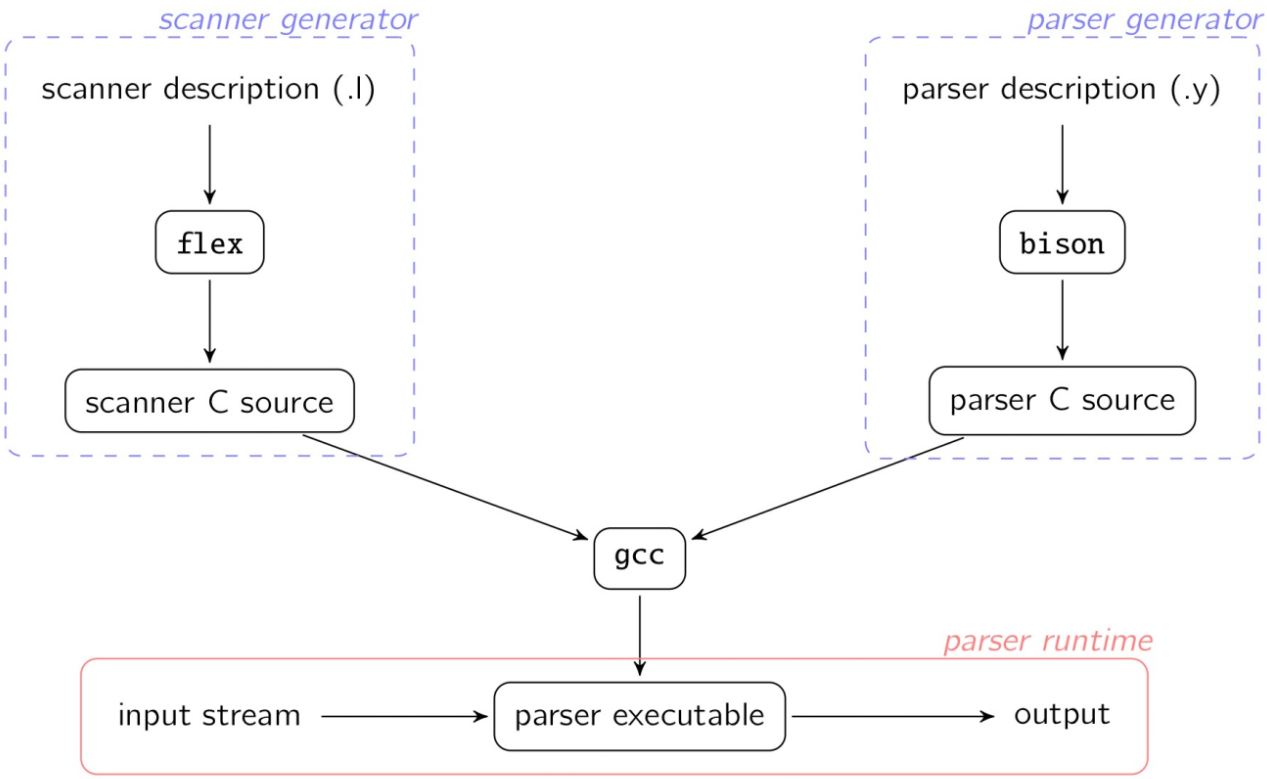
Byson uses bottom-up parsing: it always gives precedence to the inner-most rule.

```
%union{ //in the %union I list the possible semantic data types
    float f_value;
    struct Expr expr;
}

//Here I specify the type which the tokens can assume:

%token <f_value> FLOAT //%token are terminals
%type <expr> expr //%type are non terminals
```



The parser generated by Bison would be responsible for recognizing and parsing the ACSE code while the lexical analyzer (generated by Flex) would be responsible for breaking the code up into individual tokens.

8.1 ACSE



Acse is a LANCE (a simplified version of C) compiler which emits RISC-like assembly code and is built on Flex and Bison.

ACSE is a simplified compiler in order to reduce the effort to understand how compilers work. ACSE accepts a C-like source language called LANCE:

- very small subset of C99
- standard set of arithmetic/logic/comparison operators
- reduced set of control flow statements (while, do-whi le, if)
- only one scalar type (int)
- only one aggregate type (array of ints)

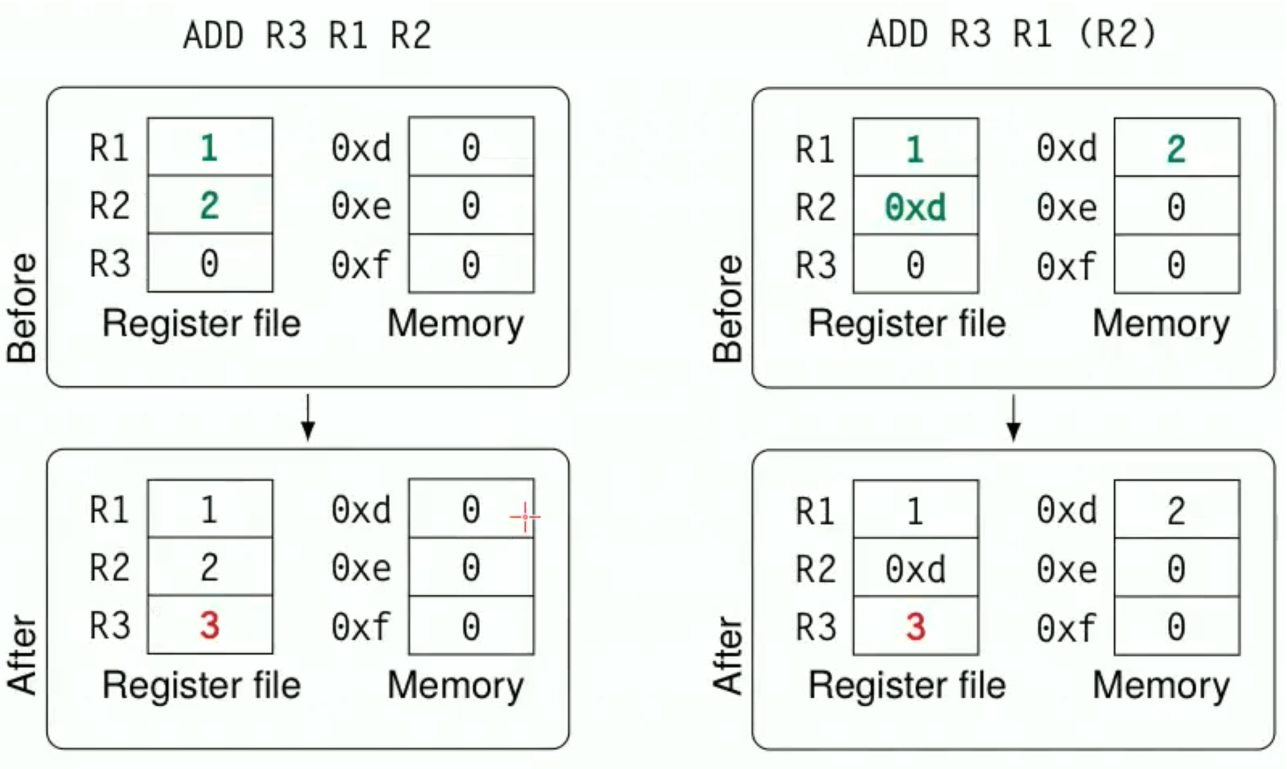
- only two I/O operations:
- `read(var)` stores into var an integer read from standard
- `write(var)` writes var to standard output writing

LANCE produces a RISC-like assembly language:

Type	Operands	
Ternary	1 destination and 2 source registers	ADD R3 R1 R2
Binary	1 destination and 1 source register, and 1 immediate operand	ADD R3 R1 #4
Unary	1 destination and 1 address operand (label)	LOAD R1 L0
Jump	1 address operand	BEQ LO

Jump instructions: - BT: unconditional branch - BEQ: branch if last result was zero - BNE: branch if last result was not zero

Using `( )` to point to the memory address.



Special registers:

- `R0` zero, always contains 0

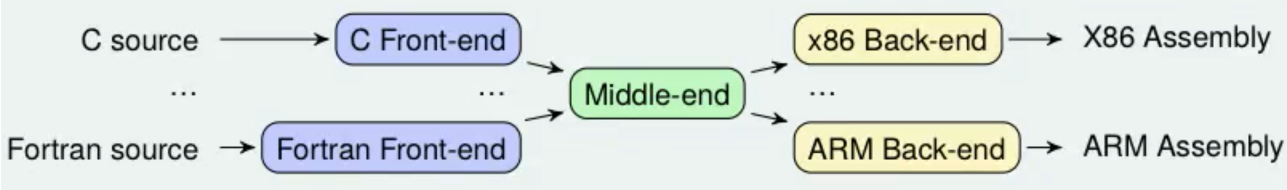
- status word `PSW` , mainly exploited by conditional jumps
  - N, negative
  - Z, zero
  - V, overflow
  - C, carry

ACSE works using **mace** which is a simulator of the fictional MACE process.

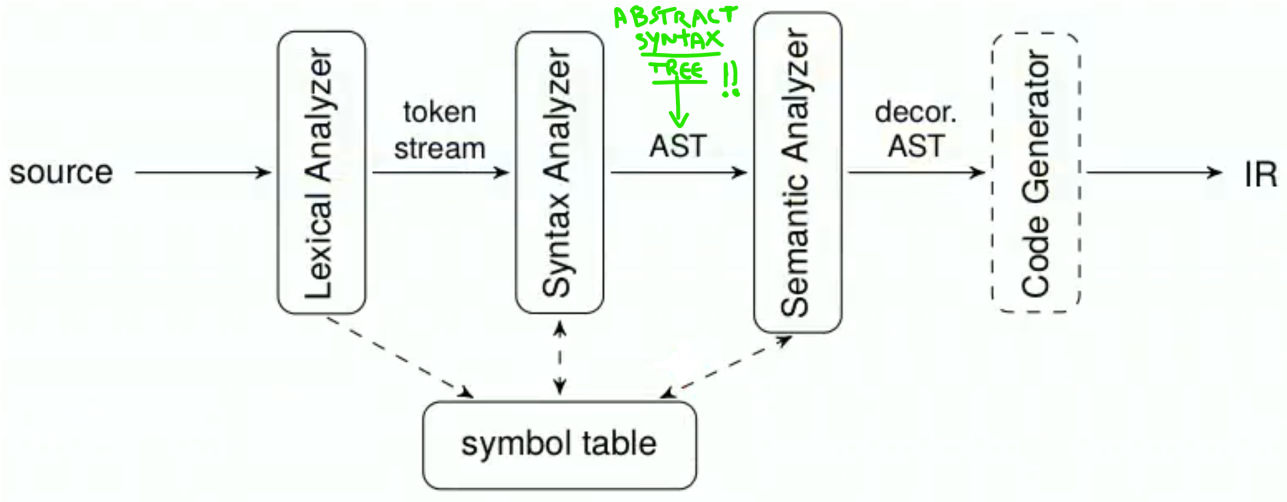
Compiler translates a program written in a language and it's organized as a pipeline:

- front-end: source language into intermediate forms
- middle-end: where transformations and optimizations are applied (for example vectorization)
- back-end
- Front-end: the source code is tokenized by a flex-generated scanner, while the stream of tokens is parsed by a bison-generated parser. At the end, the code is translated to a temporary intermediate o a representation by the semantic actions in the parser.
- No middle-end since no optimizations are made
- Back-end: we use the MACE processor

A real-world example of a compiler is LLVM.



The theory part of the course mainly focused on the frontend:



ACSE has:

- reduced set of control flow statements (while, do-while, if)
- only one scalar type (int)

- only one aggregate type (arrayofuints)
- no functions
- limited I/O (just `read(var)` and `write(var)`)

Why this language? This is an ¨academical" language and its focus to education. So there is basically nothing so that they can ask stuff to be implemented during the exam.

The parser (Bison) modifies the intermediate languages made of variable and instructions list. Then backend process it.

Precedence and associativity of expressions are handled by Bison.

Constant folding: optimization to "merge" constants at compile time.
To do this we use a structure which memorize the "value" and the type "IMMEDIATE|REGISTER". If at compile time there is a sum of two immediate, they will merged inside a single immediate.

To do this, you will use:

```
handle_bin_numeric_op()  //Arithmetic and logical operations
handle_binary_comparison()   // Comparisons
```

at the exam the grammar part is the most easy part. The difficult part is the semantic action.

Tip: It's always useful to "de-sugar" a construct you are implementing to clear up any doubt you might have about its implementation

for syntax sugar -> any for can be replaced with while

If in Bison part we use " $2" we refer to the second token

Semantic actions are independent blocks or scopes and variables declared in a semantic action are local to that action! Our exam consists in:

- add tokens modifying Flex (lexical)
- add grammar rules to recognize new constructs using Bison (syntactic)
- write semantic actions to generate code for new constructs

In particular:

1) add the "keyword" to the Flex token declaration:
   - `"keyword" { return KEYWORD_TOKEN }`
2) then in Bison (`Acse.y`) we place the token definition:
   - `%TOKEN KEYWORD_TOKEN`
3) Then you have to define the syntactic rules or modifications to existing one.
4) Define the semantic actions needed to implement the required functionality, always in `Acse.y`. General stuff in the exam could be:
   - generate a custom structure to manage results
   - use some of the 3 possible techniques to manage nesting expressions and the "stack".
   - modify existing code

resources:

- `Acse.lex` : flex source (scanner)
- `Acse.y`: Bison syntax grammar of LANCE in `Acse.y` . The semantic actions are responsible for the actual translation from LANCE to assembly
- `codegen`: instruction generation functions: `aze_gencode.h` where there are all the helper functions to generate assembly.

approach:

1) read the text

2) rewrite the text using a snippet of pseudo code
3) add the tokens needed in the `Acse.lex`

```
[0-9]+        { yylval.value = atoi(yytext);
                return X; }
```

5) add the tokens in `Acse.y` with `%token`.

```
%token <value> X
```

6) Is it a statement or an expression? You have to decide it.
7) add the "rule like a context-free grammar" in the `Acse.y` and so the **semantic action**
8) for the semantic action we probably need to define a struct in the `acse_struct.h` associated with what we are adding
9) any new struct declared as to be added as an item in the `%union{ <here> }`] ??
10) add the struct declared to the new token] ??

8.2 ACSE Cheatsheet

8.2.0.1 Basics

-

8.3 In order to use a value of an identifier non-terminal, we have to know where the variable is located (in which register). The function `get_symbol_location` is used in order to retrieve the register location assigned to a given identifier. `int reg = get_symbol_location(program, $NUM, 0);`. The returned value is not the value contained in the register but the number of the register we need to use in the instructions!

-

8.4 always free `$NUM` the identifiers at the end

- in a statement it makes no sense to use $$ because I don´t have to pass any value to the caller
-

8.5 $$ cannot be assigned in a midrule action

Check if a nonterminal (usually exp) is an immediate or a register:

```
// after this piece of code you can handle expression not caring if it is immediate or register
if ($2.expression_type == IMMEDIATE) {
    gen_addi_instruction(program, value, REG_0, $2.value);
} else {
    gen_add_instruction(program, value, REG_0, $2.value, CG_DIRECT_ALL)
}
```

- Adding another register value into a register: `gen_add_instruction(program, dest_reg, dest_reg, source_reg, CG_DIRECT_ALL);`

8.5.0.1 Immediates

- Creating a new register: `int reg = getNewRegister(program);`
- Creating a register and assigning an immediate value (in this case value assigned = 0):

```
int r_i = gen_load_immediate(program,0); //it returns registry ID
```

- Get variable from token with `getVariable(program,char * id)`:

```
t_axe_variable *v_dst = getVariable(program,$1)
t_axe_variable *v_src1 = getVariable(program,$3)
t_axe_variable *v_src2 = getVariable(program,$5)
```

- array element into a new register

```
// in case of register index
int r_i = gen_load_immediate(program,2);
//r_i can be modified here
int reg = loadArrayElement(program, $NUM, create_expression(r_i, REGISTER));
```

- alternative in case of immediate index (in this case accessing element 2):

```
int reg = loadArrayElement(program, $NUM, create_expression(2, IMMEDIATE));
```

- Saving a nonterminal array (represented by identifier) into a variable: `t_axe_variable *array = getVariable(program, $NUM);`
-

8.6 `$NUM` needs to be freed at the end of code.

- Declaring label: `t_axe_label *label = newLabel(program)`
-

8.7 Fixing label position of an already declared label in the code: `assignLabel(program, {labelname})`

- Declaring label and fixing its position in the same point (use only for backwards jumps): `t_axe_label *label = assignNewLabel(program, {labelname})` —
-

8.8 Unconditional jump to label: `gen_bt_instruction(program, {labelname}, 0)`

- Jump if `r_index` is greater than array length:

```
gen_sub_instruction(program,getNewRegister(program),r_array->isArray,r_index)
gen_ble_instruction(program, l_exit , 0);
```

Assigning label as global variable to a token: in the token declaration `%token <label> {tokenname}` then, to initialize it in the code of the rule `${num corresponding to the token} = newLabel(program)` and to place it ‘assignLabel(program, ${num corresponding to the token})

```
int reg_i = gen_load_immediate(program, 0); //i=0

t_axe_label *exit_lbl = newLabel(program) ; //declare end of the loop
t_axe_label *loop_lbl = assignNewLabel(program) ; //declare start of the loop

//LOOP CONDITION
handle _binary_comparison(program,
                          create_expression(reg_i, REGISTER),
                          create_expression(s1_array->arraySize, IMMEDIATE),
                          _LT_); // i < size
gen_beq_instruction(program, exit_lbl, 0); // in case we skip to the end of the loop

// ***************
// LOOP BODY
// ***********

gen_addi_instruction(program, reg_i, reg_i, 1); // i++
gen_bt_instruction(program, loop_lbl, 0); //branch back

assignLabel(program, exit_lbl); //here is the label to exit the loop
```

8.8.0.1 Sharing variables

We have 3 ways to share variables between semantic actions, basically equivalent:

1) global variable: super easy to apply but it doesn't work if the statement is nestable.
2) Re-purpose a symbol's semantic value as a variable: one of the token is used to store the value, generally use this at the exam. Sometimes the variable is a new struct which we make.
3) Stack method: most complicated one and sometimes overkill.

8.8.0.1.1 Global variable

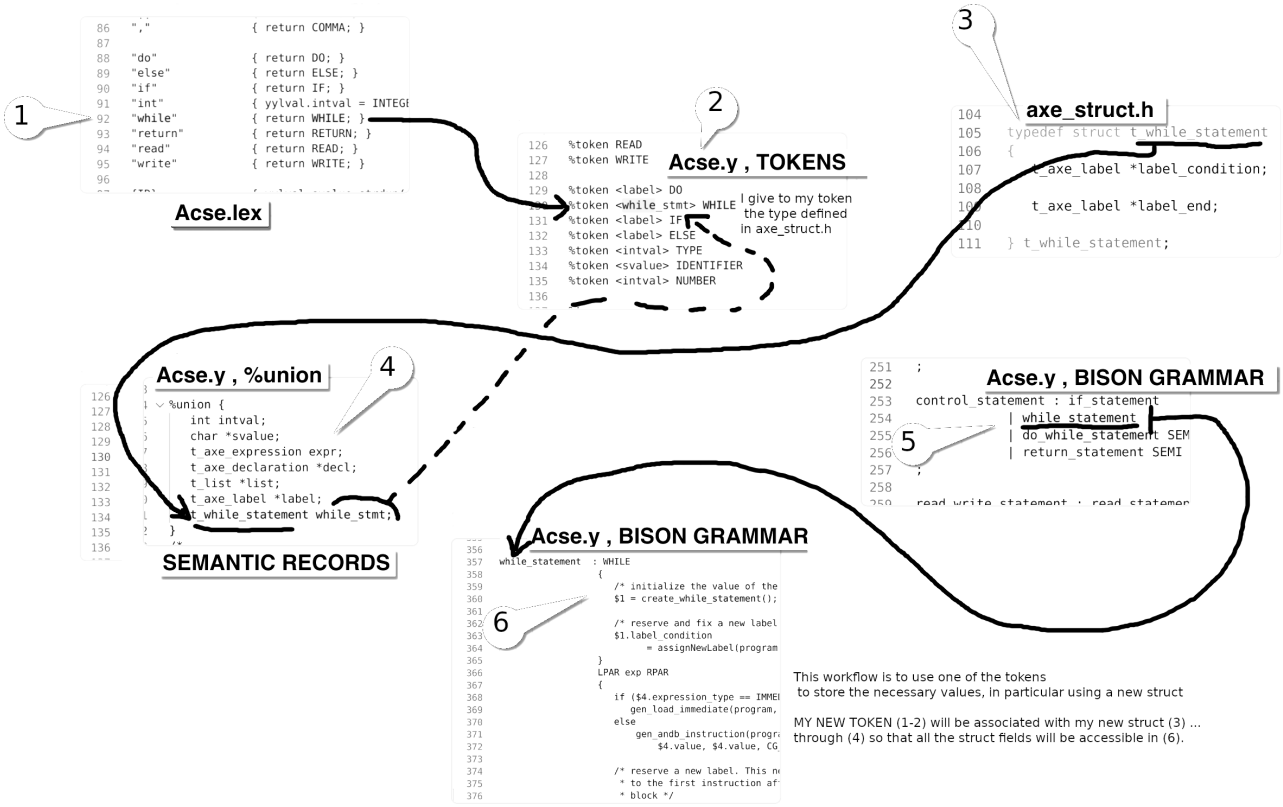
Initializing a global register: `int glob_reg;` before the "semantic records" section in Acse.y, then initialize it with `glob_reg = getNewRegister(program);` before using it.

8.8.0.1.2 Struct in (2)

To implement the 2nd option it is usually needed to declare a new structure. How to declare a new structure: - Add a new type to give global properties to a token - In the token section: `%token <struct_of_token_name> token_name` - Add to the `%union` struct: `t_new_struct struct_of_token_name` (when I create a new struct should I always include it in the `%union`). - In the axe_struct.h file:

```
typedef struct t_new_struct{
    ...members/fields...
} t_new_struct
```

- Access variables of the struct in the rule code: `$NUM.{variable name}`, where obviously you the `$NUM` is corresponding to the token name



8.8.0.1.3 List, Stack method 2nd option is the best but not possible to do in this case to make it in nestable situation where it's necessary to use "a stack method". In particularly we have to use a linked list. We have the t_list struct defined in `collections.h` as:

```
typedef struct t_list
{
    void *data;
    struct t_list *next;
    struct t_list *prev;
}t_list;
```

And we have mainly this methods:

```
/*add an element `data' to the list `list' at position `pos'. If pos is
 *negative, or is larger than the number of elements in the list, the new
 *element is added on to the end of the list. Function `addElement' returns a pointer to the new head of
extern t_list* addElement(t_list *list, void *data, int pos);

/*add an element at the beginning of the list */
extern t_list* addFirst(t_list *list, void *data);

/*remove an element at the beginning of the list */
extern t_list* removeFirst(t_list *list);
```

The generic pattern scheme is:

1) Declare in the var. declarations in acse.y the stack:

```
t_list *actual_stack = NULL;
```

2) then we can push:

```
//push
t_stack_node *top = malloc(sizeof(struct_stack_node));
top->value = get_load_immediate(program,0);
top->lbl = newLabel(program);
actual_stack = addFirst(actual_stack,top);
```

3) and pop

```
//pop
t_stack_node *top = (t_stack_node*)LDATA(actual_stack);
$$ = create_expression(top->value,REGISTER);
assignLabel(program,top->lbl);
```

where to get the data associated to the list item we use `#LDATA(item)`. 0) But we have to declare as always, in `axe_struct.h`, the struct we use to memorize the values to push to the stack.

```
typedef struct t_stack_node
{
    int value;
    t_axe_label *lbl;
} t_stack_node;
```

Example of unroll it at compile time with t_list:

```
for(int i=0, i<array->arraySize;i++){

    if(cur_list_elem ==NULL) yerror("expression list too short");

    //loading the t_axe_expression
    t_axe_expression *cur_list_elem_exp =
        (t_axe_expression *) LDATA(cur_list_elem);

    mul = handle_bin_numeric_op(program,
                                *cur_list_elem_exp,
                               create_expression(r_val,REGISTER), //an generic expression
                               MUL);

    free(cur_list_elem_exp)
    cur_list_elem = LNEXT(cur_list_elem)
}

if(cur_list_elem !=NULL) yerror("expression list too long");
```