Formal Languages And Compilers
Laboratory

Christian Rossi

Academic Year 2023-2024

Abstract

The lectures are about those topics:

- Definition of language, theory of formal languages, language operations, regular expressions, regular languages, finite deterministic and non-deterministic automata, BMC and Berry-Sethi algorithms, properties of the families of regular languages, nested lists and regular languages.
- Context-free grammars, context-free languages, syntax trees, grammar ambiguity, grammars of regular languages, properties of the families of context-free languages, main syntactic structures and limitations of the context-free languages.
- Analysis and recognition (parsing) of phrases, parsing algorithms and automata, push down automata, deterministic languages, bottom-up and recursive top-down syntactic analysis, complexity of recognition.
- Translations: syntax-driven, direct, inverse, syntactic. Transducer automata, and syntactic analysis and translation. Definition of semantics and semantic properties. Static flow analysis of programs. Semantic translation driven by syntax, semantic functions and attribute grammars, one-pass and multiple-pass computation of the attributes.

The laboratory sessions are about those topics:

- Modelisation of the lexicon and the syntax of a simple programming language (C-like).
- Design of a compiler for translation into an intermediate executable machine language (for a register-based processor).
- Use of the automated programming tools Flex and Bison for the construction of syntax-driven lexical and syntactic analyzers and translators.

Contents

CHAPTER 1

Regular expressions

1.1 Definition

Regular expressions represent a highly valuable tool for parsing. They have widespread industry support and are characterized by their simplicity, ease of composition, and considerable potency.

Definition (*Regular expression*)**.** Regular expressions constitute a grammar defining a regular language.

Definition (*Regular language*)**.** A regular language is one recognizable by a finite state automaton.

However, a notable constraint lies in the incapacity of finite state automata to perform counting operations, thereby limiting the parsing capability concerning grammars involving parentheses or similar constructs.

1.2 Regular expressions syntax

The symbols employed in regular expressions include:

Syntax	Matches
`x`	The character `x`
`.`	Any character except newline
`[x,y,z]`	Any character in the set `x, y, z`
`[^x,y,z]`	Any character not in the set `x, y, z`
`[a-z]`	Any character in the range `a-z`
`[^a-z]`	Any character not in the range `a-z`

Table 1.1: Basic character sets

Syntax	Matches
`R`	The regular expression `R`
`R S`	The concatenation of `R` and `S`
`R|S`	The alternation of `R` and `S`
`R*`	Zero or more occurrences of `R`
`R+`	One or more occurrences of `R`
`R?`	Zero or one occurrence of `R`
`R{n}`	Exactly `n` occurrences of `R`
`R{n,}`	At least `n` occurrences of `R`
`R{n,m}`	At least `n` and at most `m` occurrences of `R`

Table 1.2: Composition of regular Expressions

Syntax	Matches
`(R)`	Capture group or override precedence
`^R`	Match at the beginning of the line
`R$`	Match at the end of the line
`\t`	Tab character
`\n`	Newline character
`\w`	A word *(same as `[a-zA-Z0-9_]`)*
`\d`	A digit *(same as `[0-9]`)*
`\s`	A whitespace character *(same as `[\t\s\n]`)*
`\W`	A non-word character
`\D`	A non-digit character
`\S`	A non-whitespace character

Table 1.3: Regular expression utilities

Regular expressions are ill-suited for input validation tasks. The complexity or impossibility of certain tasks becomes apparent when relying solely on regular expressions. In such cases, a comprehensive parser is essential.

1.2.1 UNIX command line tools

The following UNIX command line tools are particularly useful for regular expressions:

- `grep -E ⟨regex⟩`
  Locates all lines in the input that match the specified regex.
- `find -E . -regex ⟨regex⟩`
  Discovers all files under the current directory whose names match the given regex.
- `sed -Ee s/ ⟨regex⟩ / ⟨replacement⟩ /g ⟨filename⟩`
  Reads `⟨filename⟩`, identifies all strings matching `⟨regex⟩`, and substitutes them with `⟨replacement⟩`. Outputs the result to the standard output.

CHAPTER 2

Flex

2.1 Introduction

FLEX conducts the lexical analysis of the input stream.

Definition (*Lexical analysis*)**.** Lexical analysis pertains to the vocabulary or words of a language, distinct from its grammar and structure.

In natural languages, words can be straightforwardly listed, but this enumeration isn't feasible for artificial languages. For instance, in the C programming language, identifiers must adhere to the following rules:

- They consist of a sequence of alphanumeric characters (including underscore _).
- They cannot commence with a digit.

Technical terms are less intricate than natural language words: their structure is simpler, they adhere to specific rules, and they typically conform to a regular language.

2.1.1 Lexical analysis

A lexical analysis is responsible for identifying tokens within a sequence of characters, such as identifiers and constants. It may also augment tokens with supplementary information like identifier names and line-by-line positions. Typically, this analysis is conducted using a scanner, although manually coding a scanner can be laborious and prone to errors. Thankfully, there are scanner generators like FLEX, which operate on regular expression descriptions to automate this process. Essentially, a scanner can be conceptualized as a comprehensive finite state automaton.

In certain scenarios, a scanner suffices as it can identify words and execute semantic actions like local transformations. However, in the context of a compiler, the scanner serves a broader purpose. It primes the input for the parser by identifying language tokens such as identifiers, constants, keywords, and punctuation. Additionally, it cleanses the input by removing comments and enriches tokens with essential information like lexical values and locations.

The workflow of FLEX is shown in the following image.

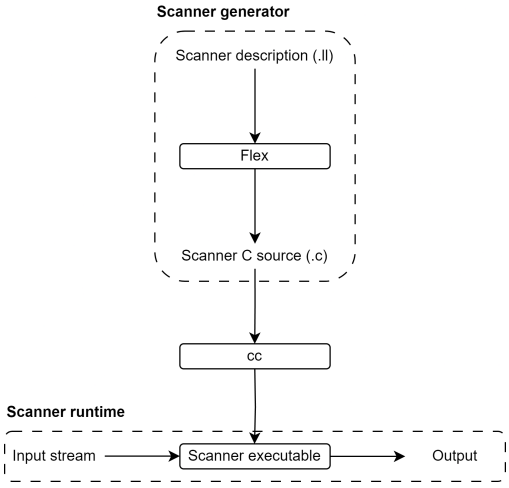

Figure 2.1: FLEX workflow

2.2 File format

A FLEX file is divided into three sections delineated by %%:

1. *Definitions*: where helpful regular expressions are declared.
2. *Rules*: which associate regular expression combinations with actions.
3. *User code*: typically containing C code, including helper functions.

2.2.1 Definitions

In lexical analysis, a definition links a name to a group of characters. Here are some key points about definitions:

- Regular expressions are utilized to define character sets.
- Literal strings are represented within quotes, and they are counted as a single symbol for precedence purposes.
- Definitions are commonly employed to define straightforward concepts, such as digits.
- They function similarly to C's preprocessor macros.
- To invoke a definition, its name is enclosed within curly braces.

Example:
Here we have some examples of definitions:

```
LETTER          [a-zA-Z$\_$]
DIGIT           [0-9]
HELLOWORLD      "*hello world*"
LETTERDIGIT     {LETTER}{DIGIT}
```

2.2.2 Rules

A rule defines a complete token to be identified. Here are the key aspects of rules:

- The token is characterized by a regular expression.
- Rules utilize definitions to create composite concepts, like numbers or identifiers.
- Each match triggers a semantic action specified within the rule.

Example:
Considering the definitions given in the previous example we can create for instance the following definitions:

```
{LETTER}({LETTER}|{DIGIT})*      {return 1;}
{DIGIT}+                         {return 2;}
[ t]+                            /* do nothing */
"if"                             {return 3;}
```

Semantic actions, executed each time a rule is matched and have access to the matched textual data. The global variables defined for semantic actions are:

- `yytext` of type `char*` that contains the matched text.
- `yyleng` of type `int` that contains the matched text.

In straightforward applications, the business logic is often embedded directly within semantic actions. However, in more intricate applications like compilers that employ a separate parser, the approach involves:

1. Assigning a value to the recognized token, known as the lexical value.
2. Returning the token type.

2.2.3 User code

The user-provided C code is directly replicated into the generated scanner without alteration. It's advantageous to include the following components:

- The main function.
- Any additional routines called by semantic actions.
- Scanner-wrapping routines.

Arbitrary code can be inserted within the definitions and rules sections by escaping from FLEX using %{, %} braces. This code is directly copied into the generated scanner without modification. It's commonly utilized for tasks such as header inclusions, defining global variables, and declaring functions.

2.3 Generated scanner

The scanner generated by FLEX is a C file called `lex.yy.c`. It exports:
`FILE *yyin = stdin;`
`int yylex(void);`
The `yylex` function parses the file `yyin` until a semantic action returns or the file ends (return value zero).

File ending FLEX necessitates the implementation of a single function: `int yywrap(void)`. This function is invoked when the file concludes, offering the chance to open another file and resume scanning from that point onward:

- Return 0 to indicate that parsing should continue.
- Return 1 to signify that parsing should cease.

If this behavior is undesired, the following line must be included in the scanner source: `%option noyywrap`.

Scanner behaviour The behavior of the scanner is governed by the following rules:

- *Longest matching rule*: if multiple matching strings are found, the rule that generates the longest match is selected.
- *First rule*: in case of multiple strings with the same length being matched, the rule listed first will be triggered.
- *Default action*: if no rules are found, the next character in the input is implicitly considered matched and printed to the output stream as is.

Scanner workflow The generated parser operates as a non-deterministic finite state automaton:

- The automaton endeavors to match all potential tokens simultaneously.
- Upon recognizing one:
    1. The associated semantic action is executed.
    2. The stream advances beyond the end of the token.
    3. The automaton resets.

In practice, the NFA is converted into a deterministic automaton using a modified version of the Berry-Sethi algorithm.

2.4 Multiple scanners

Sometimes is useful to have more than one scanner together. To facilitate the support for multiple scanners, the following methods are employed:

- Rules can be designated with the name of the associated scanner, known as the start condition.
- Special actions enable the transition between scanners.

A start condition, denoted by `S` is utilized to annotate rules as `<S>RULE` and activate rules when the scanner operates in the `S` start condition. Start conditions can be:

- *Exclusive*, declared with %`x S`; this disables unmarked rules when the scanner operates in the `S` start condition.
- *Inclusive*, declared with %`s S`; unmarked rules are active when the scanner operates in the `S` start condition.

The initial condition is inclusive by default. Additionally:

- The `*` start condition matches any start condition.
- The initial start condition is denoted as `INITIAL`.
- Start conditions are represented as integers.
- The current start condition is stored in the `YY_START` variable.

CHAPTER 3

Bison

3.1 Introduction

Definition (*Syntax analyzer*)**.** Syntax analysis, also known as parsing, involves studying the rules that dictate how words or other elements of sentence structure are combined to create grammatically correct sentences.

The syntax of a language is delineated by a grammar. Syntactic analysis entails identifying grammar structures, validating syntactic accuracy, and constructing a derivation tree for the input. Syntactic analysis processes a stream of terminal symbols, which can be categorized as follows:

- *Terminal symbols*: tokens typically generated by the lexer.
- *Nonterminal symbols*: produced solely through the reduction of grammar rules.

BISON is the primary tool used for generating syntax analyzers, and it seamlessly integrates with FLEX. Based on the LALR(1) theory, a variation of LR(1), BISON employs a push-down automaton controlled by a directed graph. The parsing stack is employed to maintain the parser's state during runtime.

The workflow of FLEX is shown in the following image.

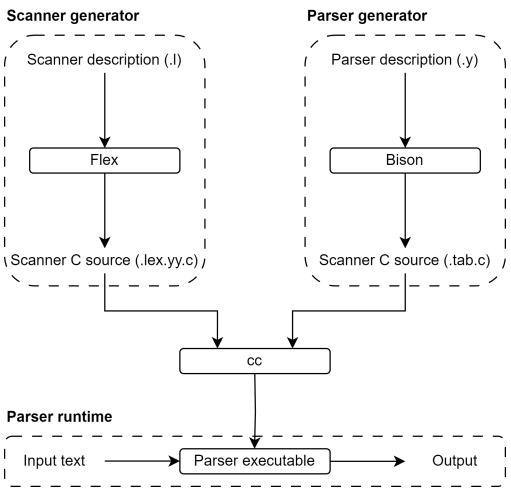

Figure 3.1: BISON workflow

3.2 File format

A BISON file is divided into four sections delineated by %%:

1. *Prologue*: this is a convenient section to include header file inclusions and declare variables.
2. *Definitions*: this section encompasses the definition of tokens, operator precedence, and non-terminal types.
3. *Rules*: this section comprises the grammar rules.
4. *User code*: typically containing C code, including helper functions.

3.2.1 Definitions

The most important part of the definitions sections are the token declarations:
`%token IF ELSE WHILE DO FOR`
BISON also generates a header file which defines a code for each token. This allows the scanner to know these codes.

3.2.2 Rules

Grammar rules are specified in Backus Normal Form notation. If not specified, the left-hand side of the first rule is the axiom.

Example:
An example of BNF grammar is as follows:

```
sections : sections section
         | /* empty */
         ;
section : LSQUARE ID RSQUARE options
        ;
options : options option
        | option
        ;
option : ID EQUALS NUMBER
       ID EQUALS STRING
       ;
```

Similar to FLEX, BISON permits the specification of semantic actions within grammar rules. A semantic action is a standard C code block that can be designated at the conclusion of each rule alternative. Semantic actions are executed when the rule they are associated with has been completely recognized. The consequence is that the order of execution of the actions is bottom-up with respect to the syntactic tree. You can also place semantic actions in the middle of a rule. In this case BISON normalizes the grammar in order to have only end-of-rule actions.

Semantic values The concept of semantic values operates as follows:

- Each parsed token or non-terminal is associated with a variable.
- For tokens, their value is assigned in the lexer.
- For non-terminals, their value is assigned in the semantic action(s) of that non-terminal.
- The value of these variables is then accessed in the rules that utilize the corresponding token or non-terminal.

The types of each semantic value are outlined in the definition section as follows:

- The %`union` declaration specifies the complete range of potential data types.
- Type specifications for terminals (tokens) are delineated in the token declaration.
- Type specifications for non-terminals are defined through special %`type` declarations.

Example:

```
%union {
    int int_val;                %token <str_value > ID
    const char *str_val;        %token <str_value > STRING
    option_t option_val;        %token <int_val > NUMBER
}                               %type <option_val > option
```

In a production, the semantic value of each grammar symbol is represented by a variable called $i, where i denotes the position of the symbol.

- $$ corresponds to the semantic value of the rule itself.
- Mid-rule actions are included in the numbering.
- However, mid-rule actions are subject to additional constraints:
  - Accessing values of symbols that appear later is prohibited.
  - Using $$* is not allowed.

Example:
Consider the following grammar:

```
section : LSQUARE
          ID
          RSQUARE
          { printf("%s", $2); }
          options
          { $$ = create_section($2, $5); }
          ;
```

We have the following semantic values:

- $$ to `section`.

- $1 to `LSQUARE`.
- $2 to `ID`.
- $3 to `RSQUARE`.
- $4 to `printf("%s", $2);` .
- $5 to `options`.

In the generated code, BISON declares the global variable `yylval`, which serves the following purpose:

- It stores the semantic value of the most recent token returned by `yylex()`.
- Its type is a union of the types declared in the BISON source.

This setup enables the scanner to assign the semantic value of a token: before returning the token value, the flex semantic action sets yylval accordingly.

3.3 Generated parser

The parser receives tokens from the scanner. It interprets the grammar derivations. Upon decoding each derivation, its associated semantic action is executed. These semantic actions generate the output of the parsing process. The executable form of the parser can be depicted as follows:

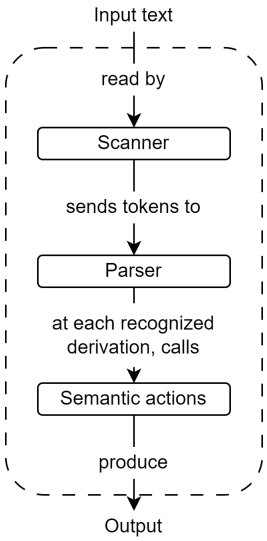

Figure 3.2: Parser executable workflow

The generated parser consists of a C file with the extension `.tab.c` and an accompanying header file with the extension `.tab.h`. The primary parsing function is:
`int yyparse(void);`
To read tokens, the parser utilizes the same `yylex()` function provided by FLEX-generated scanners. This function is invoked whenever the parser needs a new token.

3.4 Integration of flex and bison

To integrate FLEX and BISON together we have to follow these steps:

FLEX

1. Include the `*.tab.h` header generated by BISON.
2. In the semantic actions:
   - Assign the semantic value of the token (if any) to the correct member of the `yylval` variable.
   - Return the token identifiers declared in bison.

BISON

3. Declare and implement the `main()` function.

Compile time

4. Generate the flex scanner by invoking flex:
   `flex scanner.l`
5. Generate the bison parser by invoking bison:
   `bison parser.y`
6. Compile the C files produced by bison and flex together:
   `cc -o out lex.yy.c parser.tab.c`

CHAPTER 4

ACSE

4.1 Introduction

The primary objective of a compiler is to translate a program written in a source language L_0 into a semantically equivalent program expressed in a target language L_1.

A compiler is structured as a pipeline, where each stage applies a specific transformation to the input program, resulting in an output program. Each stage serves a distinct purpose:

- *Front-end*: This stage converts the source program into an intermediate form.
- *Middle-end*: here, transformations and optimizations are applied to the intermediate form.
- *Back-end*: this stage converts the optimized intermediate form into the target machine language.

4.2 Advanced compiler system for education

ACSE is a basic compiler that takes in a source language similar to C, called LANCE, and generates an assembly language resembling RISC, named MACE.

The ACSE package includes two additional auxiliary tools, completing the entire toolchain:

- Asm: an Assembler that converts assembly code into machine code.
- Mace: a Simulator for the fictional MACE processor.

The core elements of ACSE compiler are:

- Scanner: FLEX source in `Acse.lex`
- Parser: BISON source in `Acse.y`
- Codegen: instruction generation functions `axe_gencode.h`.

ACSE functions as a syntax-directed translator, meaning it generates instructions directly while parsing the source code. The sequence of compiled instructions is inherently determined by the syntax of the language. Unlike traditional compilers that initially build a syntactic tree (AST) before generating instructions, ACSE operates differently.

Its main file, `Acse.y`, houses the BISON-syntax grammar of LANCE. Within this file, semantic actions are pivotal as they handle the actual translation process from LANCE source code to assembly instructions.

4.2.1 Intermediate representation

The intermediate representation is the data representation used in a compiler to represent the program. In ACSE it is composed of two main parts: the instruction list, and the variable table.

ACSE utilizes a RISC-like intermediate assembly language that closely resembles the final output, known as the MACE assembly language. This intermediate language comprises various instructions, including:

- Arithmetic and logic instructions (e.g. `ADD`, `SUB`).
- Memory access instructions (e.g. `LOAD`, `STORE`).
- Conditional and unconditional branch instructions (e.g. `BEQ`, `BT`).
- Special I/O instructions (e.g. `READ`, `WRITE`).

Data storage in this assembly language can be achieved using either unbounded registers or unbounded memory locations.

Registers In the context of ACSE, a register identifier is represented by an integer value that denotes a specific register within an infinite bank of registers. The value of the register identifier corresponds to the number of the register it represents.

> **Example:**
> For instance:
>
> - Register R0 has the register identifier 0.
> - Register R10 has the register identifier 10.

It's crucial to emphasize that performing any arithmetic or comparison operations directly on register identifiers is incorrect.

In ACSE, there are two special registers:

- The zero register (R0) contains the constant value 0, and any writes to it are ignored.
- The status word or PSW (Program Status Word) is implicitly read from or written to by arithmetic instructions. The PSW register contains four single-bit flags, that are exploited mainly by conditional jump instructions:
  - `N`: negative.
  - `Z`: zero.

  - `V`: overflow.
  - `C`: carry.

  In ACSE, there exist fifteen conditional jump instructions, including:

- `BT`: represents an unconditional branch.
- `BEQ`: indicates a branch if the last result was zero.
- `BNE`: signifies a branch if the last result was not zero.

Each arithmetic instruction in ACSE alters the Program Status Word (PSW) based on the outcome of the computation. When encountering a branch instruction, the PSW is examined to determine whether branching should occur or not.

4.3 LANguage for Compiler Education

LANCE is the source language recognized by ACSE:

- It supports a very small subset of C99.
- It includes a standard set of arithmetic, logic, and comparison operators.
- It features a limited set of control flow statements, including while, do-while, and if.
- It comprises only one scalar type, namely int.
- It includes only one aggregate type, namely an array of ints.
- It does not support functions.

The input and output operations have very limited support:

- `read(var)` stores an integer read from standard input into `var`.
- `write(var)` writes var to standard output.

Syntax A LANCE source file comprises two sections: variable declarations and the program body, which is a list of statements. A statement is a syntactic unit of an imperative programming language that expresses some action to be carried out. Statements can be classified as:

- Simple Indivisible element of computation (e.g., assignments, read, write).
- Compound Statements which contain multiple simple statements (e.g., if, while, do-while).

Compilation Due to the absence of optimizations, there is no middle-end in the compilation process. Consequently, the compilation steps are as follows:

1. The source code is tokenized by a flex-generated scanner.
2. The stream of tokens is parsed by a bison-generated parser.
3. The code is translated to a temporary intermediate representation by the semantic actions in the parser.
4. The intermediate representation is normalized to accommodate the physical limitations of the MACE processor.
5. Each instruction is printed out, resulting in the assembly file.

4.3.1 Expressions

In the LANCE language, expressions are prevalent and support nearly all operators found in the C language, including:

- Basic arithmetic $(+, -, *, /)$.
- Bitwise operators $(\&, |, <<, >>)$.
- Logical operators $(\&\&, ||, !)$.
- Comparison operators $(! =, ==, >, <, >=, <=)$.

The expression grammar of LANCE mirrors the example infix expression grammar we've encountered during discussions about BISON. Naturally, we must declare operator precedence and associativity to properly parse expressions.

MINUS The LANCE grammar incorporates unary minus syntax for negation, denoted as `MINUS exp`. The `MINUS` operator is left associative and shares the same priority as `PLUS`, which is suitable for normal subtraction operations. However, this precedence is incorrect for negation operations.

4.3.2 Variables

In ACS every scalar variable is stored in a register. The function used to retrieve the register is `get_symbol_location` that returns the register identifier. It needs as input the name of the variable. The other information of the variables can be retrieved with `getVariable` that returns the structure `t_axe_variable`.

Registers To get a new register we can use the function `getNewRegister` that returns the identifier of the register created. To get a register with a constant we can use the function `gen_load_immediate()` function.

Helper function Another operations used to do simple operations on registers are:

- `handle_bin_numeric_op()`: for arithmetic and logical operations.
- `handle_binary_comparison()`: for comparisons.

4.3.3 Branches

In ACSE, branches can be categorized as:

- *Forward*: The label appears after the branch in the code.
- *Backward*: The label appears before the branch in the code.

Since ACSE is a syntax-directed translator, we require a mechanism to allocate a label without explicitly generating it. This task can be accomplished using the following functions:

- `newLabel()`: define a new label that will be inserted after the current instruction.
- `assignLabel()`: insert a label in the current line.
- `assingNewLabel()`.