Series 1

Introduction, compiler architecture

Grammars and Abstract syntax

Exercise 1 (Compilation errors) Give examples of code snippets (in any programming language) that produce the following errors.

1. A lexical error.
2. A syntactic error.
3. A static-semantics error.
4. A dynamic-semantics error.

Exercise 2 (From languages to grammars) We consider vocabulary $V = \{a, b, c\}$. For each of the following languages, propose a grammar.

1. $L_1 = a^* \cdot b \cdot c^*$
2. $L_2 = \{a^n \cdot b \cdot c^n \mid n \geq 0\}$
3. $L_3 = \{a^n \cdot b \cdot c^m \mid 0 < n < m\}$
4. $L_4 = \{w \mid w \text{ is a palindrome}\}$

Exercise 3 (Automata and Grammars for regular languages) We consider regular language $L_0 = a^* \cdot b^*$. Consider grammar G_0 defined by the following production rules:

$$S \rightarrow A \cdot B; \quad A \rightarrow a \cdot A \mid \epsilon; \quad B \rightarrow b \cdot B \mid \epsilon$$

where S is the axiom describing language L_0. The finite state automaton $A_0 = (Q, V, \delta, q_0, F)$ recognizes this language:

$$\begin{array}{rcl} Q & = & \{q_0, q_1\} \text{ is the set of states} \\ V & = & \{a, b\} \text{ is the input alphabet} \\ q_0 & \in & Q \text{ is the initial state} \\ F & = & \{q_1\} \text{ is the set of accepting states} \\ \delta & = & \{\ (q_0, a, q_0), (q0, \epsilon, q_1), (q_1, b, q_1)\ \} \text{ is the transition relation} \end{array}$$

1. Give a derivation of grammar G_0 for string *aaaabb*.
2. Give the sequence of transitions of automaton A_0 allowing to recognize the string *aabb*.
3. Give the sequence of configurations of automaton A_0 allowing to recognize the string *aabb*.

Exercise 4 (Context-free Languages and Grammars) We consider the context-free language $L1 = \{a^n \cdot b^n \mid n \geq 0\}$. Consider grammar $G_1 = S \rightarrow a\ S\ b \quad S \rightarrow \epsilon$ where S is the axiom describing language L_1. The push-down automaton $A_1 = (Q, V, \Gamma, \Delta, Z, q_0, F)$ recognizes this language.

$$\begin{array}{rcl}
Q & = & \{q_0, q_1, q_2, q_3\} \text{ is the set of states} \\
V & = & \{a, b\} \text{ is the input alphabet} \\
\Gamma & = & \{Z, A\} \text{ is the stack alphabet} \\
Z & \in & \Gamma \text{ is the initial stack symbol} \\
q_0 & \in & Q \text{ is the initial state} \\
F & = & \{q_3\} \text{ is the set of accepting states} \\
\Delta & = & \{\ (q_0, a, Z) \rightarrow (q_1, Z\ A), \\
 & & (q_1, a, u) \rightarrow (q_1, \omega, u\ A), \\
 & & (q_1, b, u\ A) \rightarrow (q_2, u), \\
 & & (q_2, b, u\ A) \rightarrow (q_2, u), \\
 & & (q_2, \epsilon, Z) \rightarrow (q_3, \epsilon)\ \} \\
 & & \Delta \text{ is the set of transitions, with } \omega \in V^* \text{ and } u \in \Gamma^*
\end{array}$$

1. Give the derivation of grammar G_1 for string *aabb*.
2. Give the sequence of configurations of automaton A_1 allowing to recognize string *aabb*.

Exercise 5 (Grammars for Arithmetic Expressions) We want to define a grammar purposed to describe *arithmetic expressions* built from binary operators for substraction ($-$) and multiplication ($*$) and for which operands are integers (e).

1. We first consider grammar G_0 with the following production rules:
   $Z \longrightarrow E \qquad E \longrightarrow E - E \qquad E \longrightarrow e$
   Using sequence “10 - 2 - 3”, show that this grammar is ambiguous.
2. To eliminate the ambiguity of G_0, we consider grammar G_1, defined as follows:
   $Z \longrightarrow E \qquad E \longrightarrow E - T \qquad E \longrightarrow T \qquad T \longrightarrow e$
   Draw the derivation tree corresponding to the example from the previous question. We say here that substraction is “left associative”.
3. How G_1 should be modified to introduce a multiplication operator such that:
   - multiplication is left associative;
   - multiplication has precedence over substraction.

   Justify your answer by constructing syntactic trees for the following expressions: “$10 - 2 * 3$”, “$10 * 2 - 3$”.

4. We want to allow to put a sub-expression between parenthesis so as to write e.g., “$(10 - 2) * 3$” or “$10 * (2 - 3)$”. Modify the previous grammar. Build the syntactic tree for the expression given as examples.
5. We add the unary minus operator (noted $-$), that has precedence over $-$ and $*$. Modify the previous grammar accordingly. Build the syntactic tree for “$10 * -2 - -3$”.

Exercise 6 (Grammars and Ambiguity) We consider the following grammar that describes statements from a programming language:

$$\begin{array}{lll} Z & \longrightarrow & I \\ I & \longrightarrow & \text{ if } e \text{ then } I \\ I & \longrightarrow & \text{ if } e \text{ then } I \text{ else } I \\ I & \longrightarrow & a \end{array}$$

1. Show that this grammar is ambiguous: find a sentence from the language that leads to two different derivation trees.
2. Propose a solution to make this grammar non-abiguous (by for instance modifying the described language).
3. Is it possible to make this grammar non-ambiguous without modifying the described language?

Exercise 7 (From Derivation Trees to Machine code) Let G be a grammar describing assignments where terminals `c` and `i` designate an integer constant and an identifier, respectively.

```
Z -> A ;              E -> E + T | T        F -> c | i | (E)
A -> i := E           T -> T * F | F
```

We consider the two following assignements where `x` and `y` are identifiers:
(1) $x := 5 + x * 2 + y;$ (2) $y := (5 + x) * 2 + y;$

1. Build the derivations trees corresponding to assignements (1) and (2).
2. We consider a processor with registers noted `Ri` and with the following instruction set:
   - `LD Ri,op` loads the value of `op` in register `Ri`, `op` denotes a variable or a constant,
   - `ST Ri,x` gives the value contained in register `Ri` to variable `x`,
   - `ADD Ri,op1,op2` puts the sum `op1+op2` in register `Ri`,
   - `MULT Ri,op1,op2` puts the product `op1*op2` in register `Ri`,
     where `op1` denotes a register and `op2` denotes a register or a constant.

   Write the sequence of instructions corresponding to the traduction of (1) and (2).
3. We want to generate code for the machine from derivation trees. In the sequence proposed in the previous question, to which node should the instructions of the machine be associated with?
   Decorate the derivation trees of (1) and (2) with the sequences obtained: each instruction should decorate a node from the tree, certain nodes should not be decorated.
4. Inspiring from the result of question 3, draw simplified versions of trees (1) and (2). A simplified tree is an abstract tree containing only decorated nodes.
5. Propose an abstract syntax for G (in the form of a grammar). Propose several ways to write this grammar.

Exercise 8 (Automata and Grammars) The following code snippets give examples of “for” statements in ADA and C :

```
for i in 1..N loop             for (i=0 ; i<N ; i++) {
   -- statements               /* statements */
end loop ;                     }
```

1. Propose a concrete syntax (even a simplified one) for statement ”for” for each language.
2. Propose a corresponding abstract syntax.

A Simple Compiler

Introduction

The logical steps of a compiler usually consists of:

- Lexical Analysis
- Syntactic Analysis
- Semantic Analysis
- Intermediate Code Generation
- Code Optimization
- (Target) Code Generation.

A language, its grammar

Program	::=	Block
Block	::=	`begin` Declaration_list ; Statement_list `end`
Declaration_list	::=	Declaration_list ; Declaration | Declaration
Declaration	::=	`Idf` := Expression : Type
Type	::=	`integer`
Statement_list	::=	Statement_List ; Statement | Statement

Statement	::=	`Idf` := Expression | `skip`
		| `if` Expression `then` Statement `else` Statement `endif`
		| `if` Expression `then` Statement `endif` |
		`while` Expression `do` Statement `done`

Expression	::=	Disjunction
Disjunction	::=	Conjunction | Disjunction `or` Conjunction
Conjunction	::=	Comparison | Conjunction `and` Comparison
Comparison	::=	Relation | Relation `=` Relation
Relation	::=	Sum | Sum < Sum
Sum	::=	Term | Sum + Term | Sum − Term
Term	::=	Factor | Factor ∗ Term
Factor	::=	`Not` Factor| Denotation | `Idf` | '('Expression')'

Abstract Grammar

$$\begin{array}{lll} S & ::= & x := a \mid \texttt{skip} \mid S;S \mid \texttt{if}\ b\ \texttt{then}\ S\ \texttt{else}\ S \mid \texttt{while}\ b\ \texttt{do}\ S \\ a & ::= & n \mid a+a \mid a-a \mid a*a \mid x \\ b & := & \texttt{true} \mid \texttt{false} \mid a=a \mid a \leq a \mid \neg b \mid b \wedge b \end{array}$$

Lexical Analysis

A scanner (lexical analyzer) has as input a character string and outputs a pair lexical class, element of the class.

Exercise 9 (Lexical Analysis) Determine the lexical classes and give them a specification as an automaton or a regular expression.

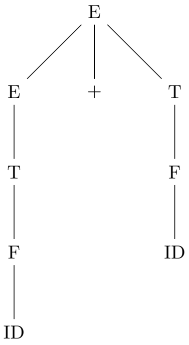
Syntactic Analysis

A parser (syntactic analyzer) has as input a flow of lexical classes and outputs an abstract tree described by the abstract grammar. The syntactic tree is obtained by rule derivation.of the abstract grammar. The abstract tree is obtained from the derivation tree: it should contain operations as nodes and IDs as leaves. When reducing, the derivation tree to a syntax tree, terminals are "lifted" in the tree. You will use the following grammar for expressions:

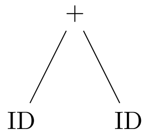
$$\begin{array}{ll} E & \rightarrow E + T \mid E - T \mid T \\ T & \rightarrow T * F \mid F \\ F & \rightarrow \mathit{ID} \mid \mathit{NUM} \end{array}$$

For instance, for the expression $a + b$, we get:

Syntactic tree



Abstract tree



Exercise 10 (Derivation trees and AST)

Build the derivation trees and the abstract syntax trees corresponding to the expressions below.

1. $a + b - c$
2. $a - b + c$
3. $a + b * c$

Type checking

Type checking is performed on the abstract tree.

Exercise 11 (Rules for type checking)

1. Give informally the rules for type checking. Rules should be given on the abstract grammar.

Code Generation

As input to code generation, we have an abstract tree. As output, we have a sequence of statements. Considering the abstract syntax of Section 1, we define the following generation functions:

- A function that takes as input a statement S, and that produces a sequence of machine statements.
- A function that takes an arithmetical expression, and that produces a sequence of machine statements and the register that contains the result.
- A function that takes as input a Boolean expression and that produces a sequence of machine statements and the register that contains the result.

The M Machine

In the following, we describe a machine with registers. The arithmetical and logical operations update the condition codes. The arithmetical and logical operations, noted `OPER`, are `ADD, SUB, AND, OR`.
General registers are noted `Ri`. The `PC` register designates the program counter, the register `FP` designates the basis of local environment, and the register `SP` the head of the stack. Statements and addresses are coded on 4 bytes, the stack pointer designates the last occupied slot. We push on the stack by decrementing by 4 the stack pointer et we pop on the stack by incrementing by 4 the register `SP`. Integers are encoded on 4 bytes.
To each variable of a program is associated an address of the form `FP - shift`, where `shift` is computed and inserted in the symbol table.
In this exercise, we are interested in elementary operations, which syntax is given below. We will see branching operations and procedure calls later in lecture course on code generation.

`OPER` Ri, Rj, Rk	Ri is the destination register
`OPER` Ri, Rk, val	Ri is the destination register
`LD` Ri, [addr]	
`ST` Ri, [addr]	

Addresses are described as follows:
addr ::= Ri + Rj | Ri + val | Ri | val

Exercise 12 (Generating 3-address code) Give the sequence of code that corresponds to the following expressions or sequences of statements. We suppose having the following symbol table.

identifier	shifting
x	4
y	8
z	12
u	16
v	20
w	24

1. x-y+z
2. x := 3
3. x := 3 ; y := x+2 ;
4. x := 3 ; y :=2 ; z:= x + y
5. x := 3 ; y := 2 ; z := x + y u := x + y+ z ; v := z + y ; w := y + v ;

Local optimizations

Exercise 13 (Optimizing code) Minimize the number of registers used in the previous code sequences. To this purpose, we can be interested in:

- the minimal number of register to evaluate an expression;
- the analysis of active variables to minimize the `LD` and `ST` operations that can be avoided;
- a bound of the total number of registers and the backup policy.

The backup policy is in place when the number of registers is not sufficient in order to perform the computation. In that case, the value of a register can be “backed up” in the local environment (after the variables of the local environment). That is for the symbol table mentioned below, the first free slot is located at [FP-28].

Global Optimizations independent from the machine

Exercise 14 (global Optimizations) Let us consider the following program written in an intermediate representation:

```
(1) a := 1
(2) c := a+b
(3) if c>0 goto 7
(4) d := a+b
(5) c := c+2
(6) goto 9
(7) f := a+b
(8) c := c-1
(9) g := d+c
```

1. Build the control-flow graph.
2. Suppress redundant computations.
3. Suppress the assignments that can be avoided, while supposing that:
    - no variable is used outside the code snippet,
    - *g* is used outside the code snippet.
4. Constant propagation.