Mid-term exam of compilation

M1 Informatique Fondamentale, ENS Lyon
Vendredi 22 novembre 2013

Duration: 2 hours. Course notes are authorized.
This exam is made of 3 independant exercises and 1 appendix.
Exercise 2 must be answered on a separate sheet.

Exercise 1. *Lexical analysis*

On the alphabet $\Sigma = \{+, ., 0, 1, e\}$, we consider the **lexical description** with the following tokens:

- **Token -1-:** $+$
- **Token -2-:** $(0|1)^+$, *example: 101*
- **Token -3-:** $1.(0|1)^+e(0|1)+$, *example: 1.0e10*

The generator of lexical analyzers seen in labs produces the automaton given in appendix.

Q1) In a few sentences, explain how this automaton is produced.

Q2) Detail the execution of the lexical analyzer on the following inputs:

- 1.+$
- 10+1.01e10$

$ is a token which denots the end of the input flow (EOF).

Exercise 2. *Syntactic analysis*

For this exercise, write your answers on a separate sheet.
Let us consider the following grammar:

$$\begin{array}{lcl} S & \rightarrow & T \mid I \\ T & \rightarrow & \texttt{ID} \mid \texttt{-}\ T \\ I & \rightarrow & \texttt{INT} \mid \texttt{-}\ I \end{array}$$

Where `ID` is any string, `INT` any integer and "`-`" the minus symbol.

Q3) Is this grammar $LL(k)$? For which k, if any?

Q4) Gives an equivalent grammar that is $LL(1)$.

Sometimes placing actions in the $LL(1)$ grammar would be fastidious. Therefore, to avoid reworking the grammar we want to actually parse using the original one.

The following figure depicts the automaton generated by $ANTLR$[1] for the non-terminal S (T and I behaves like for the $LL(1)$ parser).

Q5) What is the meaning of the transitions exiting q_2? Try it on "`-----1`".

Q6) If the input never contains "`--`" this automaton behaves like a $LL(k)$ parser. For which k?

[1] A widely used $LL(*)$ parser generator

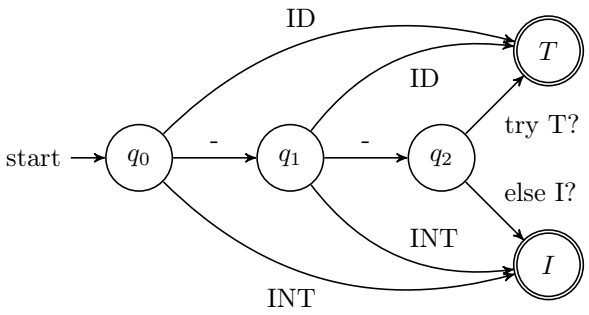

Figure 1: Automata with backtracking for non-terminal S

Exercise 3. *Syntax-directed translation*

In C, local variables are necessarily declared in the main block of the function body (see the left program below). We want to declare local variables in any sub-block (see the right program) with the usual scoping rules:

- The *scope* of a variable declaration (part of code where the variable exists) in a block is limitated to that block and its sub-blocks.
- If an instruction is in the scope of several declarations using the same identifier (right program, variable `x`), we consider the deepest declaration (right program, variable `x` of the sub-block).

```
int foo(int n)
{ //Main block
  int[3] t;
  int i,x;
  ...
}
```

```
int foo(int n)
{ //Main block
  int[3] t;
  int i,x;
  for(i=0; i<N; i++)
  { //Sub-block
    int[2] x;
    ...
  }
}
```

Q7) Draw the activation record for `foo`, left program.

Q8) Give the translation rule for a sub-block $[\![$ { type_1id_1; ...; type_nid_n; BODY } $]\!]_\rho$. One will use recursively $[\![$ BODY $]\!]_{\rho'}$ on an environment ρ' properly defined.

Q9) Draw the activation record right after entering the `for` block, right program.

Appendix.