Open book exam. Duration: 1h30m First Midterm Exam ("Mini-Teste")

Group 1. *First*, *Follow* Sets and *Parsers* LL(k) (5 pts)

Consider the CFG[1] below.

```
S → AT
A → aAa | bAb | #T
T → aT | bT | ε
```

1.a) [1pt] Give the *First* and *Follow* sets for each grammar variable;

Answer:

First(S)={a, b, #}

First(A)={a, b, #}

First(T)={a, b, ε} OR First(T)={a, b}

Follow(S)={}

Follow(A)={a, b}

Follow(T)={a, b}

1.b) [2pts] Show the table for the parser LL(1);

Answer:

	a	b	#
S	S → AT	S → AT	S → AT
A	A → aAa	A → bAb	A → #T
T	T → aT T → ε	T → bT T → ε	

Note that:

T → ε

Follow (T) = {a, b, $}

1.c) [2pts] Indicate the possible problems this grammar may have and that need to be solved in order it can be implemented as a top-down recursive parser LL(1).

Answer:

The LL(1) parser table has cells with more than one production (conflicts).

One problem is the fact that the grammar is ambiguous. An example showing the ambiguity:

[1] *Context-Free Grammar*

Example #aa :

$S \Rightarrow AT \Rightarrow \#TT \Rightarrow \#aTT \Rightarrow \#aaTT$
$\Rightarrow \#aa\underline{\varepsilon}T \Rightarrow \#aa\underline{\varepsilon\varepsilon}$

$S \Rightarrow AT \Rightarrow \#TT \Rightarrow \#\varepsilon T \Rightarrow \#aT \Rightarrow$
$\Rightarrow \#aaT \Rightarrow \#aa$

So, one of the first modification can be to eliminate the ambiguity.

Group 2. Lexical and Syntactic Analysis (6 pts)

We intend to design and implement a programming language. The presented CFG G1 below represents the current version of the grammar for the language.

Some of the Tokens for G1:
IDENTIFIER = [a-zA-Z][0-9a-zA-Z]*
WHILE = while
ENDWHILE = endwhile
DO = do
IF = if
ELSE = else
THEN=then
ENDIF = endif

2.a) [0.5pt] Considering that OP represents the arithmetic operations, -, +, *, /, and CMP the comparison operations, !=, ==, >, <, >=, <=, show the definitions of these tokens as regular expressions.

Answer:

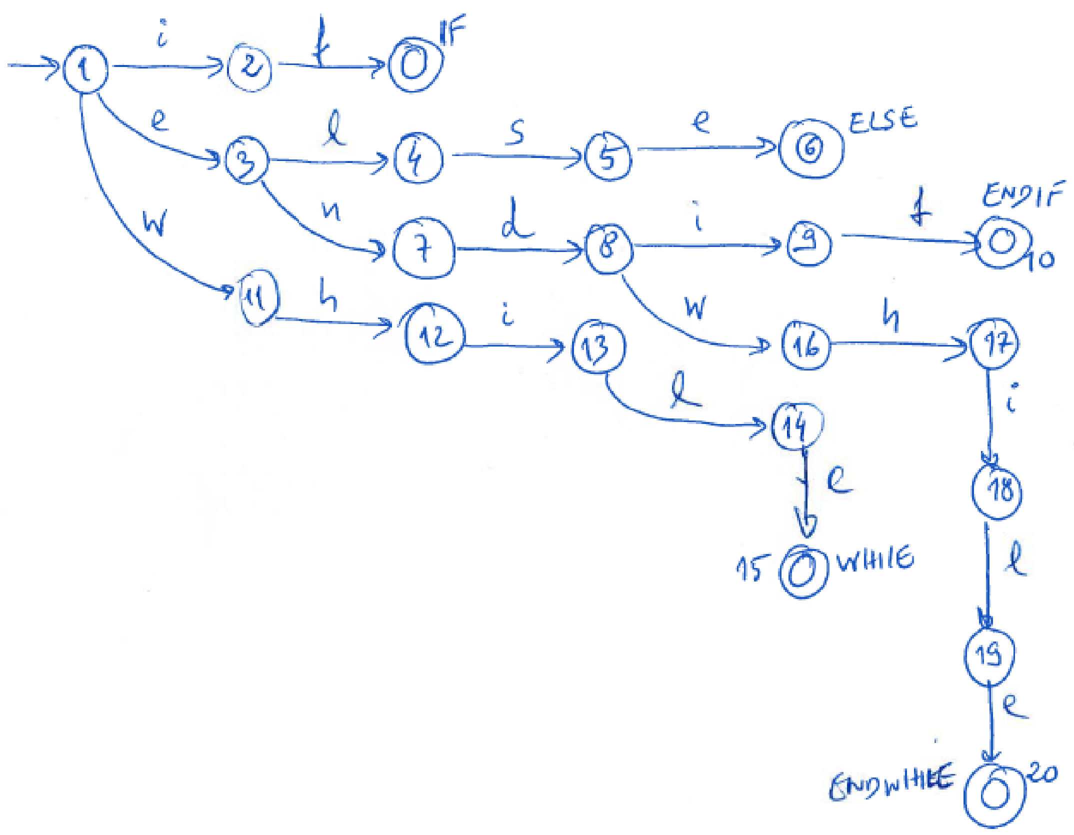
*OP = + | - | * | /*

CMP = != | == | < | > | >= | <=

2.b) [1pt] Show the DFA for the tokens WHILE, ENDWHILE, IF, ELSE, and ENDIF.

Answer:

DFA (incomplete):

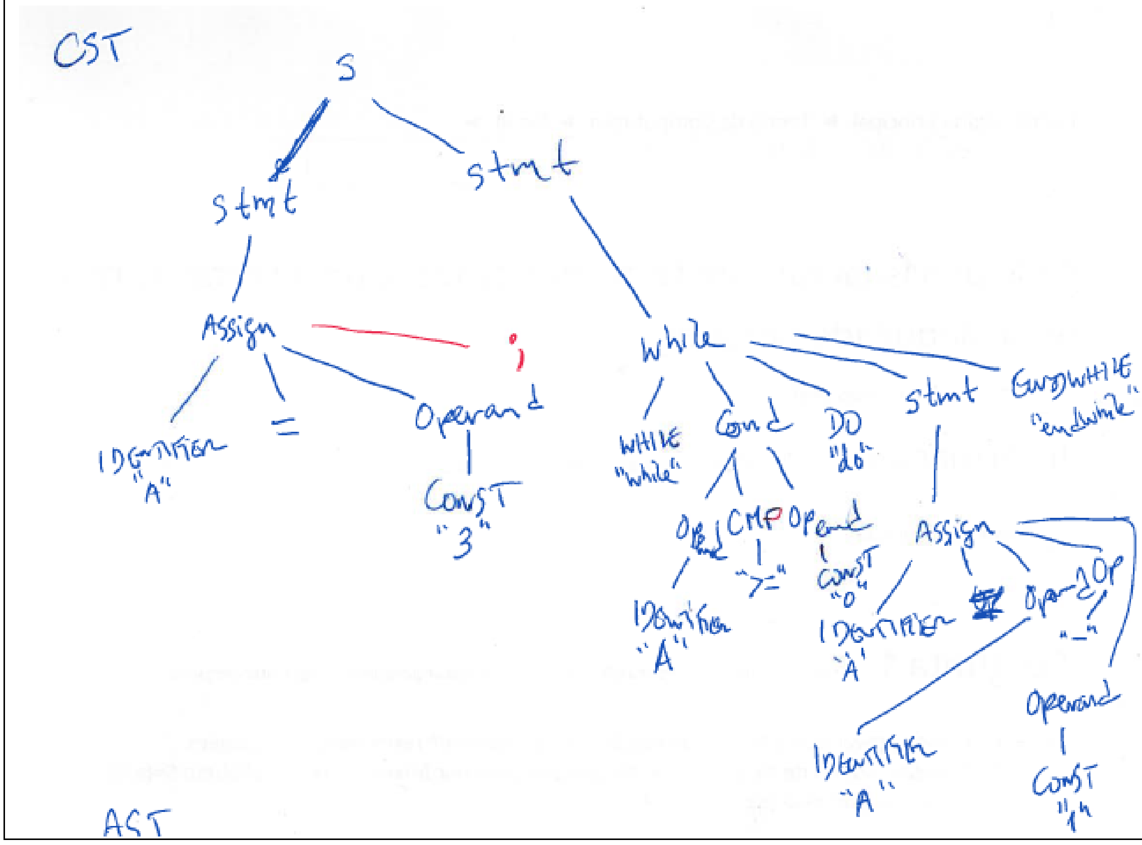


Note: the dead state is not included and is implicit.

2.c) [1pt] Show the concrete syntax trees for the code example below and considering the depicted CFG.

Answer:

CST:

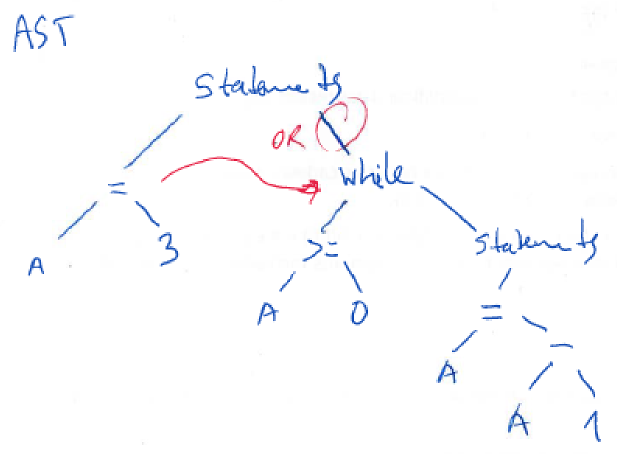


2.d) [0.5pt] Show a possible abstract syntax tree (AST[2]) for the concrete syntax tree of the previous question.

Grammar G1:
1. S → (Stmt)*
2. Stmt → While | Assign | If
3. While → WHILE Cond DO (Stmt)* ENDWHILE
4. Assign → IDENTIFIER = Operand (OP Operand)? ;
5. Operand → IDENTIFIER | CONST
6. If → IF Cond THEN Stmt (ELSE Stmt)? ENDIF
7. Cond → Operand CMP Operand

Answer:

Possible AST:



Code example:
```
A=3;
while A >= 0 do
  A = A – 1;
endwhile
```

2.e) [2pt] Show the necessary functions, considering the grammar rules of lines 2 to 3, and the respective pseudo-code to implement them, as a top-down recursive LL(1) parser,. Assume the existence of the lexical analyzer which outputs the sequence of tokens, the existence the global variable *token*, and the function *next()* which returns the next token in the sequence of tokens (in the beginning the variable *token* identifies the first token in the sequence).

[2] *Abstract Syntax Tree*

Answer:

```
// Stmt → While | Assign | If
bool Stmt() {  // all of the three have ≠ First sets
  if(token == WHILE) return While();
  if(token == ASSIGN) return Assign();
  if(token == IF) return If();
  return false;
}
// While → WHILE Cond DO (Stmt)* ENDWHILE
bool While() {
   if(token == WHILE) {
     token = next();
     if(Cond()) {
        if(token == DO) {
           token = next();
           while(token != ENDWHILE) if(!Stmt()) return false;
           token = next(); // token is ENDWHILE
           return true;
        } else return false;
     } else return false;
  } else return false;
}
```

2.f) [1pt] Suppose we want to modify the grammar in order to make possible to input values from the keyboard and print values of variable to the screen. Present the grammar modifications you suggest.

Answer:

Stmt → While | Assign | If | Read | Print

Read → cin >> IDENTIFIER ;

Print → cout << IDENTIFIER ;

(more advanced will allow the use of endl and of more than one >> or << as in C++)

Group 3. Semantic Analysis (5 pts)

Consider the segment of Java code of the example below.

Example

```
public int f1(int B[]) {
      int[] A = {1, 2, 3, 4};
      int S = 0;
      int i;
      for (i = 0; i < A.length; i++) {
         S += A[i]*B[i];
      }
      return S;
 }
```

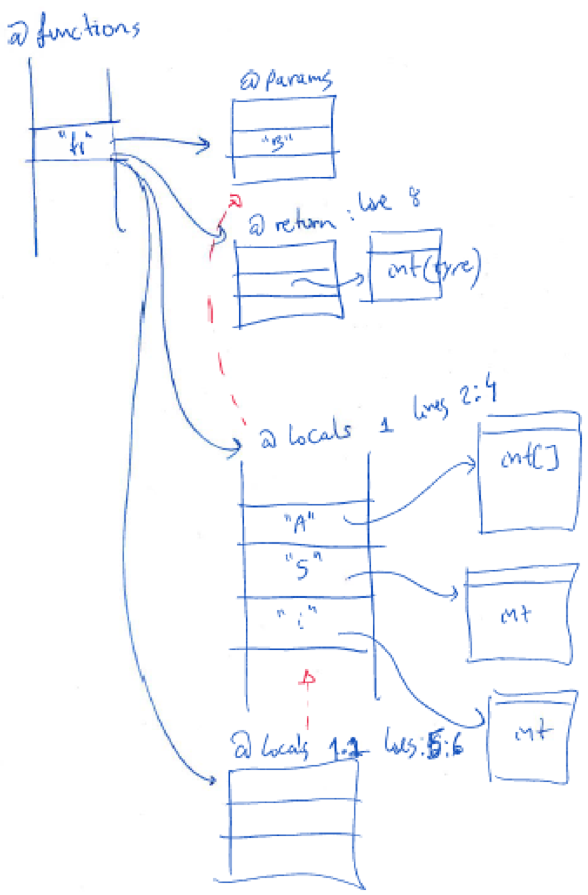
3.a) [2pts] Assuming that the variables used in the code can only be parameters or local variables, show a possible symbol table for this example indicating the information to be included in the descriptors.

Answer:

Considering the following scopes:

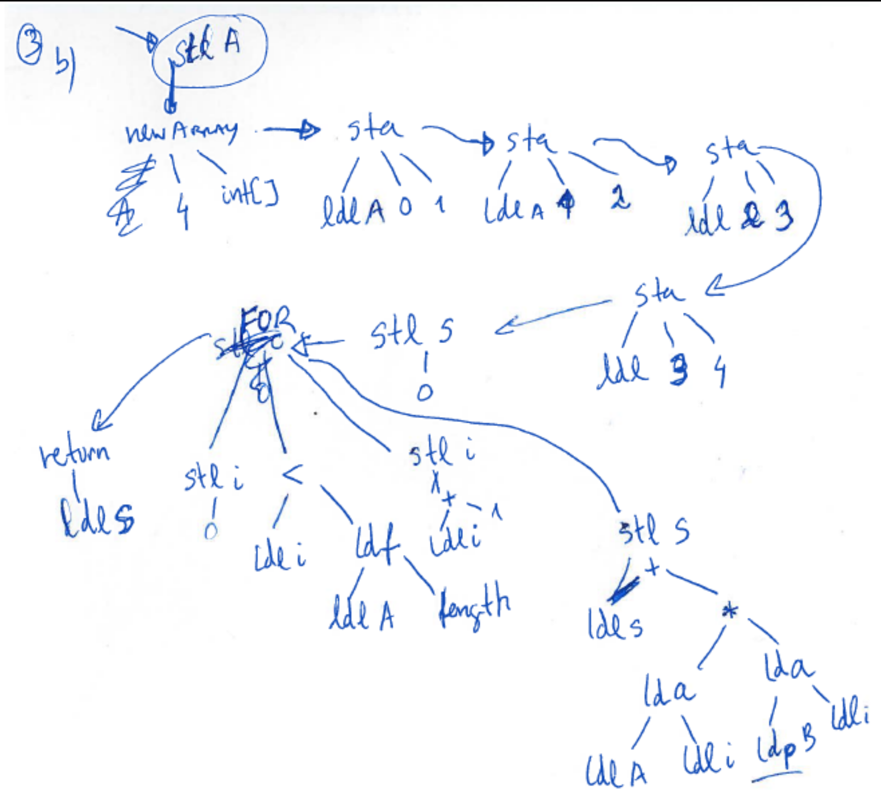
```
@function
public int f1(int B[]) { @params
    int[] A = {1, 2, 3, 4}; @1
    int S = 0; @1
    int i; @1
    for (i = 0; i < A.length; i++) { @1.1
        S += A[i]*B[i]; @1.1
    }
    return S; @1
}
```

Possible Symbol Table:



3.b) [2pts] Show a high-level intermediate representation for the body of the function, using as reference the high-level intermediate representation (HIR) based on expression trees and presented in the classes of the course.

Answer:



3.c) [1pt] In some programming languages, such as Java, the indexing of arrays neither can have negative values nor values greater than N-1 (with N the size of the specific array dimension being indexed). Discuss the inclusion of this kind of verification in the semantic analysis of a compiler, mentioning the possible problems and solutions.

Answer:

This kind of verification is impossible to do for all cases at compile time as the indexing of an array variable might depend on the inputs of the program. There are simple cases where the identification might be easier such as when the indexing is an expressions of constants. However, as these might be very simple cases also easily identified by the programmer, most compiler do not do this analysis. In the case of Java, it is the JVM that checks the values of the indexing of arrays at runtime.

Note: Other possibilities would be to use an analysis based on range values in order to understand what is the range of the expression for a particular array indexing.

Group 4. General (4 pts)

Comment the following sentences, indicating if they are true or false, and justifying your answers (if helpful include illustrative examples to help on justifying your answers):

4.a) [2pts] "The existence of ambiguity in a given CFG always implies conflicts in its LL(1) parser table.".

Answer:

TRUE. If the grammar is ambiguous it means that there are at least two possible derivations accepting at least one string. This would reflect in the LL(1) parser table with at least two productions in one of the cells.

4.b) [2pts] "Compilers split the lexical, syntactic and semantic analysis in separated stages because it is impossible to implement them in a single stage.".

Answer:

FALSE. All of them can be merged as the lexical analysis can be implemented with CFGs and the semantic analysis can be performed during the parsing steps. The reason to not implement them all together is the more complex, less efficient, and less modular, solution needed when merging all the three analysis.

(End.)