Primeira Prova de Compiladores – 2011.2.

Aluno:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. (2,0) Identifique quais das tarefas abaixo se refere à análise léxica (L), sintática (S), às duas (LS), ou deixe em branco se não for realizada por nenhuma das duas fases.

( ) Verifica se uma variável foi declarada antes de ser utilizada

( LS ) Faz parte do front-end de um compilador

( C ) Verifica se o número de parâmetros passados em uma chamada a um método está correto

( ) Verifica se o tipo dos parâmetros passados em uma chamada a um método está correto

( S ) Verifica se o “else” aparece sempre depois de um “if”

( LS ) Interage com a tabela de símbolos (lê ou escreve)

( S ) É especificada utilizando gramáticas livres de contexto (BNF / EBNF)

( L ) Lê o arquivo de entrada e gera os tokens

( S ) Cria a abstract syntax tree na memória

( S ) Recebe os tokens como entrada

1. (2,0) Nao se pode garantir **estaticamente** que todos os acessos a arrays são feitos dentro dos limites do tamanho do array. Explique que tipos de erro podem ser capturados pelo lexer, parser, analisador semantico, e os que são checados apenas dinamicamente (como no exemplo acima).

Lexer: escrever um caracter não conhecido na linguagem. Digamos que a linguagem não tenha “£” e o array foi definido como array[£];

Parser: não obedecer a regra da gramática, como em array [else], enquanto deveria ser array[ID] | array [NUMBER];

Semântico: não obedecer o tipo, números ou ordem dos parâmetros, como no caso arry[x,y] enquanto só recebe um parâmetro.

Os que são checados dinamicamente são: se é tentando acessar um elemento de um não array ou se é tentando acessar um elemento fora do tamanho do array.

**Dada a gramática abaixo, responda as questões seguintes**

Program : Program ";" Program ***Program***

| Statement ***SimpleStmt***

;

Statement : "if" Expression "then" Statement "else" Statement ***IfStatement***

| IDENTIFIER "=" Expression ***AssignmentStatement***

;

Expression : IDENTIFIER ***Id***

| NUMBER ***Literal***

| IDENTIFIER "+" Expression ***Add1***

| NUMBER "+" Expression ***Add2***

;

1. (1.0) A gramática acima possui recursão à esquerda? Explique e em caso afirmativo indique em qual ou quais produções. Em caso positivo, apresente uma versão sem recursão à esquerda.

Sim, há uma recursão à esquerda na produção Program : Program “;” Program.

Há duas possíveis soluções para esse problema:

1. Program : "if" Expression "then" Statement "else" Statement ";" Program

| IDENTIFIER "=" Expression ";" Program

2. Program : Statement ";" Program

1. (1.0) A gramática acima é ambígua? Explique, e em caso afirmativo indique em qual ou quais produções. Em caso positivo, apresente uma versão sem ambiguidades.

Sim, pois por haver a produção à esquerda em Program ";" Program é possível gerar, por exemplo, Statement “;" Statement ";"Statement derivando a regra a esquerda ou a direita. Sendo assim, qualquer uma das duas versões acima resolvem a ambiguidade.

1. (1.0) Descreva como pode ser feita a implementação de um recursive descent parser para a gramática acima. Apresente e justifique modificações na gramática se forem necessárias.

Para implementar um recursive descendent parser não podemos deixar a recursão à esquerda, sendo assim usarei o exemplo 2 para criar o recursive descendent parser:

void Program (){

switch (lookahead) {

case Statement (): Statement (); if (lookahead != null) match (TO\_SEMI\_COL); if (lookahead != null) Statement (); break;

default: syntax\_error();

}

}

void Statement (){

switch (lookahead) {

case (TO\_IF): match (TO\_IF); Expression (); match (TO\_THEN); Statement (); match (TO\_ELSE); Statement (); break;

case (TO\_IDENTIFIER): match (TO\_IDENTIFIER); match (TO\_EQUAL); Expression (); break;

default: syntax\_error();

}

}

void Expression (){

switch (lookahead) {

case (TO\_IDENTIFIER): match (TO\_IDENTIFIER); if (lookahead != null) match (TO\_PLUS); if (lookahead != null) Expression (); break;

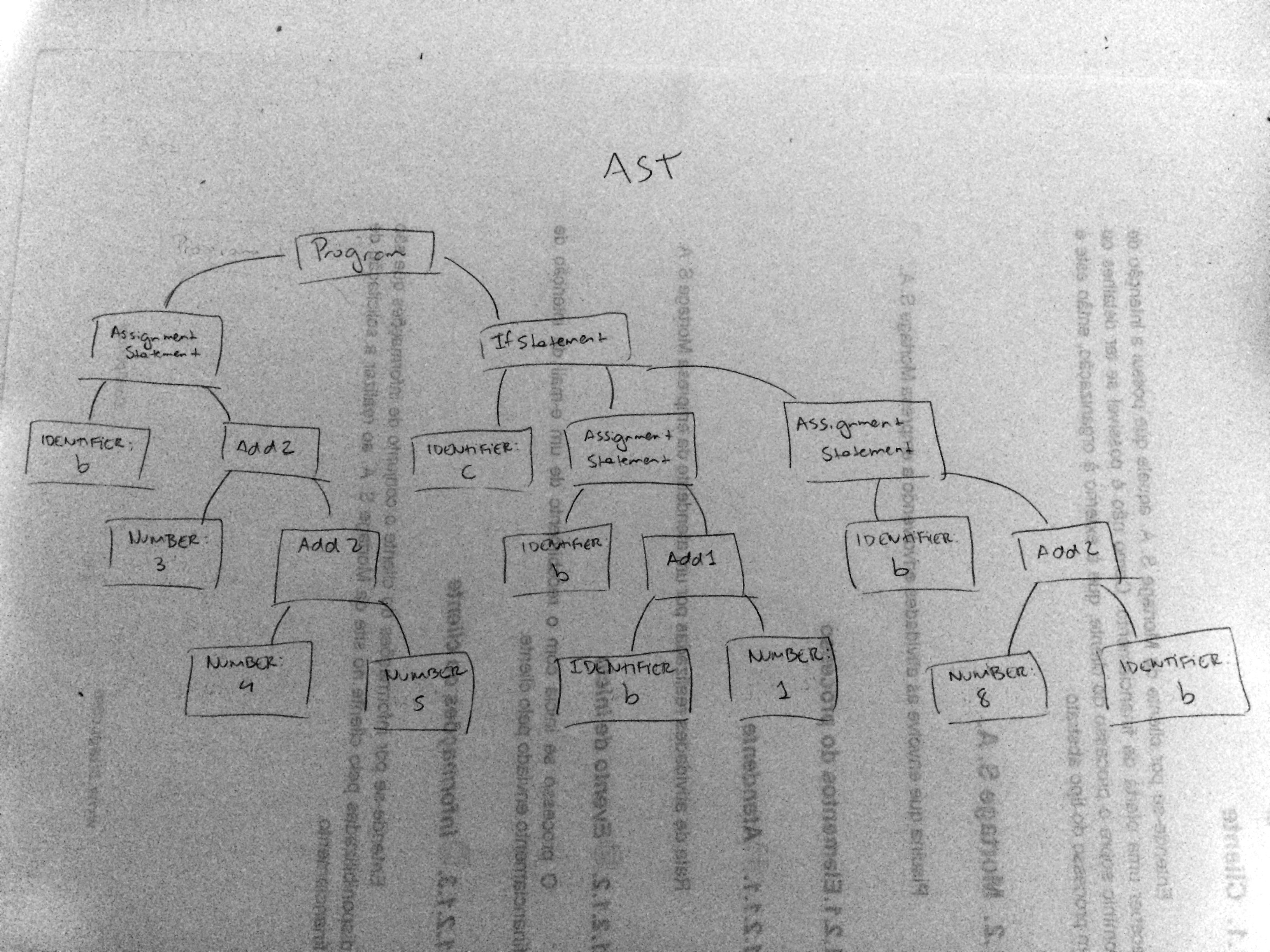
case (TO\_NUMBER): match (TO\_NUMBER); if (lookahead != null) match (TO\_PLUS); if (lookahead != null) Expression (); break;

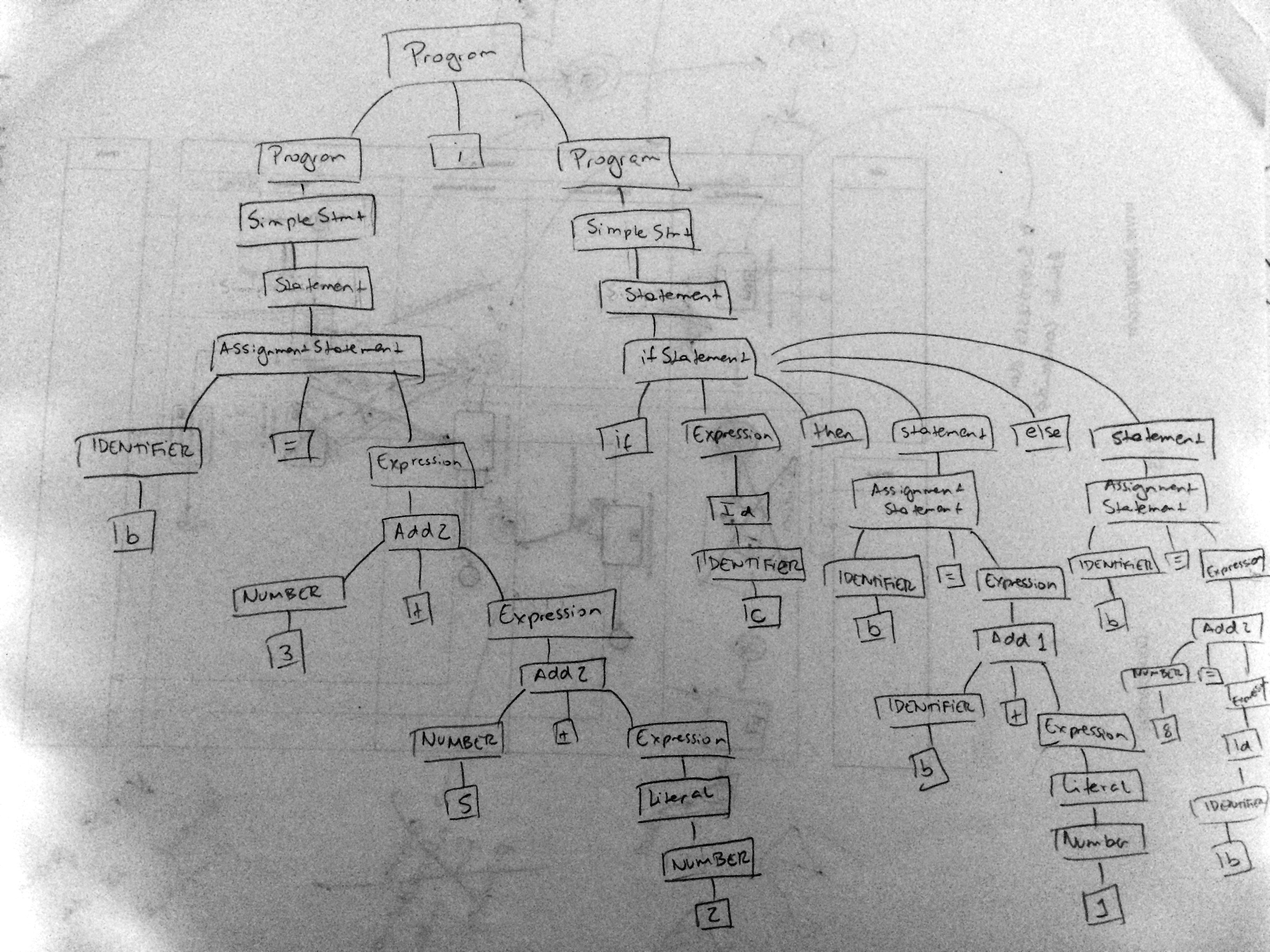
default: syntax\_error();

}

}

1. (1.0) Usando a gramática acima, sem modificações, desenhe uma árvore sintática abstrata e uma parse tree para o programa abaixo., identificando qual é a AST e qual a parse tree.

**b = 3 + 4 + 5; if c then b = b + 1 else b = 8 + b**

****

1. (1.0) Mostre que modificacoes você faria na gramatica para considerar expressoes booleanas.

Alteraria a regra Expression para o seguinte:

Program : Program ";" Program ***Program***

| Statement ***SimpleStmt***

;

Statement : "if" Expression "then" Statement "else" Statement ***IfStatement***

| IDENTIFIER "=" Expression ***AssignmentStatement***

;

Expression : IDENTIFIER ***Id***

| NUMBER ***Literal***

| IDENTIFIER "+" Expression ***Add1***

| NUMBER "+" Expression ***Add2***

| BoolExpression ***Bool***

;

BoolExpression : Expression "&&" Expression ***And***

| Expression "||" Expression ***Or***

| "!" Expression ***Not***

| Expression "==" Expression ***Equals***

| Expression "!=" Expression ***NotEquals***

;

1. (1.0) Mostre que modificacoes você faria na gramatica para permitir que regras basicas no uso de tipos fossem checadas estaticamente e indique como você faria para implementar estas chacagens.

Program : Program ";" Program ***Program***

| Statement ***SimpleStmt***

;

Statement : "if" Expression "then" Statement "else" Statement ***IfStatement***

| CHAR "=" Expression ***AssignmentStatement***

;

Expression : CHAR ***Id***

| INT ***Literal***

| CHAR "+" Expression ***Add1***

| INT "+" Expression ***Add2***

;

INT : [0-9]+ ;

CHAR : [a-zA-Z];