Midterm Exam 2

601.428/628 Compilers and Interpreters

November 8, 2021

Complete all questions.

Time: 75 minutes.

I affirm that I have completed this exam without unauthorized assistance from any person, materials, or device.

Signed: _____

Print name: _____

Date: _____

Question 1. [25 points] Consider the following context-free grammar rules, each of which has an attribute grammar rule for computing an attribute called **par**. The nonterminal symbols are $\boxed{\text{L B}}$, and the terminal symbols are $\boxed{0\ 1}$. L is the start symbol.

Production	Attribute grammar rule
$L \rightarrow B$	L.par $\leftarrow$ B.par

$L_0 \rightarrow L_1\ B$	if (B.par = 0)
	  L_0.par $\leftarrow L_1$.par
	else
	  L_0.par $\leftarrow$ (L_1.par + 1) mod 2

$B \rightarrow 0$	B.par $\leftarrow$ 0
$B \rightarrow 1$	B.par $\leftarrow$ 1

The **par** attribute is "parity". An input string with an even number of occurrences of 1 has parity 0, and an input string with an odd number of occurrences of 1 has parity 1. Note that 1 mod 2 = 1 and 2 mod 2 = 0.

Note that the subscripts (e.g., the 0 in L_0) are only used to distinguish occurrences of the same nonterminal symbol in a production.

(a) Show a derivation for the string $\boxed{1101}$.

Working string	Production
$\underline{L}$	$L \rightarrow$

(b) Show a parse tree for the derivation you found in (a). Annotate each node in the parse tree to show the value of the **par** attribute. For example, if a node for the B nonterminal has a child which is the terminal symbol 1, its **par** attribute value should be 1.

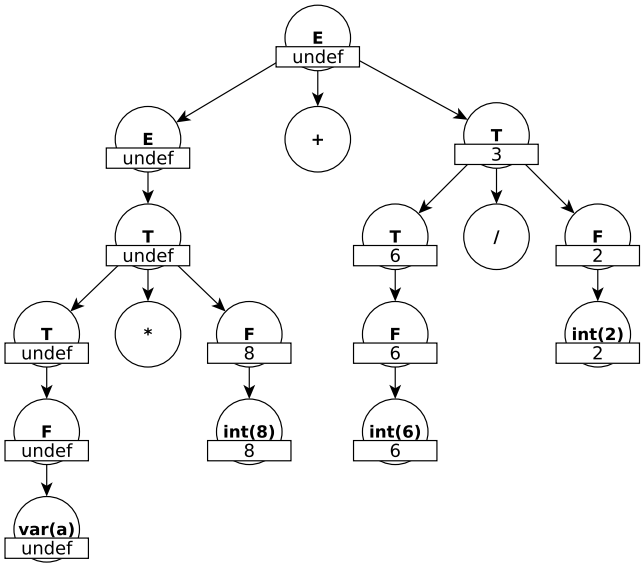
Question 2. [25 points] Consider the following context-free grammar with nonterminal symbols E T F and terminal symbols + - * / **int var** (E is the start symbol):

E → T	T → T * F
E → E + T	T → T / F
E → E - T	F → **int**
T → F	F → **var**

Assume that **int** indicates an integer literal specified as a sequence of one or more decimal digits and **var** indicates a variable name specified as a sequence of one or more lower case letters (a–z).

For each production, specify attribute grammar rules defining a **cv** attribute. The **cv** attribute value should be the constant value of the expression, or a special *undef* value if the expression's value is not constant. An expression is constant if its value does not depend on any variable references.

As an example, the expression a * 8 + 6 / 2 would have the parse tree shown below, with boxes showing the expected value of the **cv** attribute for each node:



Recall that attribute grammar rules must be *functional*: they compute an attribute value based on attribute values of child, parent, and/or sibling nodes. There is no global state or mutable state in an attribute grammar computation. If you need to use one or more additional attributes (besides **cv**), you may. You will need to specify rules to fully compute all attributes.

You may assume that a `str_to_int` function exists which will convert a **int** terminal symbol's lexeme to an integer value.

[Specify your attribute grammar rules on the next page.]

Production	Attribute grammar rule(s)
$E \rightarrow T$	
$E_0 \rightarrow E_1 + T$	
$E_0 \rightarrow E_1 - T$	
$T \rightarrow F$	
$T_0 \rightarrow T_1 * F$	
$T_0 \rightarrow T_1 \ / \ F$	
$F \rightarrow$ **int**	
$F \rightarrow$ **var**	

Question 3. [25 points] The program on the right uses the same source language as Assignments 3 and 4.

(a) Sketch the contents of each symbol table required to fully represent semantic information about the declarations in the program. For each symbol table entry, be sure to indicate the name (identifier), kind (variable, constant, or type), and type representation. If a symbol table has a parent, indicate which symbol table is the parent.

Hint: the representation of a record type could include a pointer to the symbol table in which its fields are defined.

```
PROGRAM q3;
  CONST N = 3;
  TYPE Point = RECORD
    x, y: INTEGER;
  END;
  VAR a : INTEGER;
  VAR p : Point;
BEGIN
  a := N;
  p.x := a;
  p.y := 4;
  WRITE p.x;
  WRITE p.y;
END.
```

(b) For each identifier used in the instructions of the program (between `BEGIN` and `END`), state which symbol table entry the identifier refers to. (You can draw arrows from the code to your symbol tables in part (a), or you could give each symbol table entry a unique ID, and label each program identifier with the appropriate ID.)

Question 4. [25 points] Consider the following infix expression language:

- The operators are `+ - * /`
- Primary expressions are integer literals and "registers"
- Each register holds an integer value
- Parentheses can be used to control order of evaluation

As an example, the expression `(r1 + 3) * 4` would compute the sum of the value in the r1 register and 3, and then multiply that sum by 4.

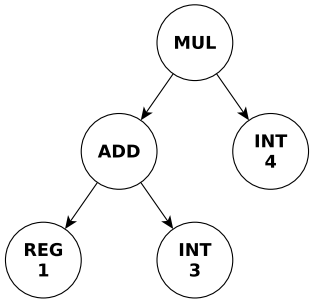
The following C++ data types could be used to define AST nodes for this language:

```
enum ASTTag { ADD, SUB, MUL, DIV, INT, REG };

struct ASTNode {
  ASTTag tag;             // what kind of AST node is it
  int ival;               // register number or integer value
  ASTNode *left, *right;
  ASTNode(ASTTag tag_, int ival_=0)
    : tag(tag_), ival(ival_), left(nullptr), right(nullptr)
    { }
};
```

For example, the AST for the expression shown above could be constructed as follows (the AST structure is shown on the right):

```
ASTNode *ast = new ASTNode(MUL);
ast->left = new ASTNode(ADD);
ast->left->left = new ASTNode(REG, 1);
ast->left->right = new ASTNode(INT, 3);
ast->right = new ASTNode(INT, 4);
```



Let's assume that a compiler will generate code for a stack-based assembly language with the following instructions:

Instruction	Meaning
pushr *R*	Push value of register *R* onto stack
pushi *N*	Push integer value *N* onto stack
add	Pop right operand, pop left operand, add, push sum
sub	Pop right operand, pop left operand, subtract, push difference
mul	Pop right operand, pop left operand, multiply, push product
div	Pop right operand, pop left operand, divide, push quotient

[Question continues on next page.]

One possible translation of the example expression is the following code:

```
pushr 1
pushi 3
add
pushi 4
mul
```

The idea is that the code generated to evaluate an expression should leave a single value — the result of evaluating the expression — on the stack.

Complete the following function. It should take a pointer to an AST node representing an expression, and print (to `stdout` or `cout`) a sequence of stack instructions to evaluate the expression.

Hints:

- Use recursion
- You may assume that nodes with `ADD`, `SUB`, `MUL`, and `DIV` tags have exactly two children

```
void codegen(ASTNode *ast) {
```

[Extra page for answers and/or scratch work.]