Lab session 3a: Compiler construction
(Bison/Yacc) parser + scanner (Flex) for Mini Pascal

The third lab session of the course *Compiler Construction* consists of two parts: part3a (this document) and part 3b. The deadline for lab 3 (both components) is Friday Dec. 22nd, 23:59.

Part 3a consists of one exercise: making a parser + scanner that accepts a program written in the programming language `Mini Pascal`. This small programming language is a miniature subset of the programming language Pascal.

The parser for `Mini Pascal` must be made using `Bison`, and the scanner must be made with `Flex`. The grammar of `Mini Pascal` is included as an appendix to this document. Note that keywords (which are terminals of the grammar) are written in boldface letters (e.g. the keyword **while**). Moreover, some non-keyword terminals are also given in boldface. They are listed in the following table.

boldface token	description
id	Case sensitive identifiers: starts with a letter, possibly followed by letters and digits.
num	Number, can be a real number or an integer
assignop	The assignment operator is written in Pascal as :=
relop	comparison operators: can be `<`, `<=`, `>`, `>=` or `<>` (latter means not equal)
addop	Addition operator: can be `+` or `-`
mulop	multiplication operators: can be `*`, `/`, `div`, or `mod`. The latter two are only for integers.

Submitting your work to Themis:
You are expected to submit a `tar`-file to Themis, which contains all the files needed to build your parser. You need to include a `Makefile`, since Themis will build your parser using the command `make`. Your code will be tested by Themis. Your program must produce the output `ACCEPTED` for valid input (i.e. a syntactically correct miniPas program). For incorrect input, your parser should produce `PARSE ERROR (%d)`, where `%d` is replacecd by the input line number where the error was encountered. After error reporting, your program should stop (terminate). Of course, a real frontend of a compiler would produce much more informative error messages. However, since we use Themis as a assessment tool, we have to restrict ourselves to minimal error reporting. Feel free to make a better version once you passed Themis' test cases.

input:

```
{ Pascal implementation of Euclid's algorithm.
}

PROGRAM euclid (input, output);

FUNCTION gcd_recursive(u, v : integer) : integer;
BEGIN
  IF u mod v <> 0 THEN
    gcd_recursive := gcd_recursive(v, u mod v)
  ELSE
    gcd_recursive := v
END;

{ main program starts here }
BEGIN
  readln(a, b);
  writeln(gcd_recursive(a,b))
END.
```

output:

```
ACCEPTED
```

input:

```
PROGRAM euclid (input, output);

function gcd_iterative(u, v : integer) : ;
var t : integer;
begin
  while v <> 0 do
  begin
    t := u;
    u := v;
    v := t mod v
  end;
  gcd_iterative := abs(u)
end;

BEGIN
  readln(a, b);
  writeln(gcd_iterative(a,b))
END.
```

output:

```
PARSE ERROR (3)
```

Mini PASCAL Grammar

program →
    program id (*identifier_list*) ;
    declarations
    subprogram_declarations
    compound_statement
    .

identifier_list →
    id
    | *identifier_list* , **id**

declarations →
    declarations **var** *identifier_list* : *type* ;
    | ϵ

type →
    standard_type
    | **array** [**num** .. **num**] **of** *standard_type*

standard_type →
    integer
    | **real**

subprogram_declarations →
    subprogram_declarations subprogram_declarion ;
    | ϵ

subprogram_declaration →
    subprogram_head declarations compound_statement

subprogram_head →
    function id *arguments* : *standard_type* ;
    | **procedure id** *arguments* ;

arguments →
    (*parameter_list*)
    | ϵ

parameter_list →
    identifier_list : *type*
    | *parameter_list* ; *identifier_list* : *type*

compound_statement →
    begin
    optional_statements
    end

optional_statements →
    statement_list
    | ϵ

statement_list →
    statement
    | *statement_list* **;** *statement*

statement →
    variable **assignop** *expression*
    | *procedure_statement*
    | *compound_statement*
    | **if** *expression* **then** *statement* **else** *statement*
    | **while** *expression* **do** *statement*

variable →
    id
    | **id** [*expression*]

procedure_statement →
    id
    | **id** (*expression_list*)

expression_list →
    expression
    | *expression_list* **,** *expression*

expression →
    simple_expression
    | *simple_expression* **relop** *simple_expression*

simple_expression →
    term
    | *sign term*
    | *simple_expression* **addop** *term*

term →
    factor
    | *term* **mulop** *factor*

factor →
    id
    | **id** (*expression_list*)
    | **num**
    | (*expression*)
    | **not** *factor*

sign →
    + | −