**An-Najah National University**
**Department of Computer Engineering**
**Compiler Construction-10636416**
**Summer 2023**
**Programming Assignment #4**
**Parser**
**Dr. Raed Alqadi**

---

In this programming assignment, you will write a parser for the ***N23*** Language. This parser reads a stream of tokens from the scanner you wrote for **PA2** and it produces abstract syntax trees (ASTs) for valid ***N23*** programs. The parser will detect and report all syntactic and some semantic errors. The ASTs will be used in later assignments as the input to the semantic checker module and for code generation.

## 1 Task #1: Produce a LL(l) Grammar

The grammar below describes the syntax of the ***N23*** language. It is not ***LL(1).*** Your first task is to produce an equivalent grammar that is ***LL(1).*** The grammar below also does not encode the precedence or associativity of operators. Your grammar should correctly capture operators' precedence and associativity. All operators are left associative. Their precedence is (from highest to lowest):

| | | | | | |
|---|---|---|---|---|---|
| *not* | *(unary)-* | | | | |
| * | / | | | | |
| + | - | | | | |
| = | != | < | <= | > | >= |
| *and* | *or* | | | | |

Operators at the same level have equal precedence. Parentheses can override the evaluation order that would result from following these rules.

**The language grammar is:**

*<program>* → *<decl_list>*

*<decl_list>* → *<decl>* *;'* *<decl_list>*
  | *λ*

*< decl>* → ***var*** ***id*** ***':'*** *<type>*
  | ***constant*** ***id*** *' =' < expr>*
  | ***function*** ***id*** *<formal_list>* ***':'*** *<type> <block>*
  | ***procedure*** ***id*** *<formal_list> <block>*

*<type>* → ***integer***
  **|** ***boolean***
  **|** ***string***

```
    <formal_list>  →      '( )'
                          | '(' <formals> ')'

    <formals>      →      id ':'  <type>
                          | <formals>  ',' id  ':'  <type>

    <stmt>         →      id ' :='  <expr>
                          | if <expr> then <stmt> fi
                          | if <expr> then <stmt> else <stmt> fi
                          | while <expr> do <stmt> od
                          | for id ' :=' <expr> to <expr> do <stmt> od
                          | read'('id')'
                          | write' ('id ')'
                          | id <arg-list>
                          | return '(' <expr>' )'
                          | <block>

    < block>       →      begin <var _decl_list> <stmt _list> end

<var _decl_list>   →      <var _decl>  ';' < var _decl_list>
                          | λ

    <var _decl>    →       var  id ':' <type>

    <stmt_list>    →       <stmt>  ';'  <stmt_list>
                          | λ

    <arg_list>      →     '('  ')'
                          | '(' <args> ')'

    <args>          →     <expr>
                          | <expr>  ','  <args>

    <expr>         →      id
                          | id <arg_lst>
                          | integer_costant
                          | string_constant
                          |< expr> <arith_op> <expr>
                          | true
                          | false
                          | <expr> <rel_op> <expr>
                          |<expr> <rel_conj> <expr>
                          | <unary_op> (<expr>
                          | '(' <expr> ')'
```

*<arith_op>* → '*' | '/' | '+' | '-'

*<rel_op>* → '=' | '!=' | '<' | '<=' | '>' | '>='

*<rel_conj>* → ***and*** | ***or***
*< unary _op>* → **-** | ***not***

Make sure that your scanner recognizes the following keywords: **function, procedure, return, not, begin**, and **end**.

## 2 Task #2: Modify the Symbol Table

Your next task is to modify the symbol table that you wrote for PA3 to accommodate a language with scoping rules similar to C's. You should add the following functions to the symbol table interface:

- *void enter_scope (..) /*Inform the symbol table that the parser is entering a new scope */*
- *void exit_scope (..) /*Inform the symbol table that the parser is leaving a scope.*/*

You may use any reasonable technique to implement a scoped symbol table.
The outer scope corresponds to global variables defined outside of any routine. Each procedure or function formal parameter list and begin-end block introduces a new scope. A variable reference uses the variable from the closest enclosing scope or the global variable. Multiple declarations in a scope are an error. ***N23*** does not have static variables. You will also need to add new fields to your symbol table entries to record the type of variables, the value of constants, and the formal parameters and result type of function. To use the AST module provided (see below), you will need to define the following functions:

- *int ste_const_value (SymbolTableEntry *e ) /*Return the value of the constant recorded in entry e.*/*
- *char *ste_name (SymbolTableEntry *e) /*Return the name of the object recorded in entry e*/*
- *j_type ste_var_type (SymbolTableEntry *e) /*Return the type of the variable recorded in entry e */*

## 3 Task #3: Write a Parser

Your next task (which comprises most of this assignment) is to write a recursive-descent parser for ***N23*** using the grammar from Task #1. The parser class should have the following public function:

- *AST *parse (FileDescriptor *fd, SymbolTable *tabl)*
- *//or AST *parse(), if the arguments are included in the constructor*

The parser scans the input from the file descriptor (fd) and records information on identifiers in the symbol table (tbl). The parser should parse one declaration from the top level of the file and return its AST. To parse the next item, the parser is invoked again. If the parser detects an error, it should print an error message and return an empty tree, i.e., NULL. The parser should return a distinctive node on end-of-file.
The parser should detect all syntax errors. It should also detect the following semantic errors: Multiple declarations of a variable, procedure, or function.
.Use of a variable, procedure, or function before it is declared.
The parser may quit after detecting and reporting the first error. You should use the routine *report_error* from ***PA2*** to print error messages. The file *parser.h* defines the interface.

## 4 Support Routines

To simplify this assignment, I am providing an abstract datatype for the abstract syntax trees. You need to modify the code to be compatible with C++.
The files *ast.h*, *types.h*, and *ast.c* contain the code for ASTs.1
The function:

- *AST *make_ast-.node (AST_type type, ...) /*creates and returns a new AST node. Its first argument is the type of the node. The rest of the arguments are initial values for all fields in nodes of that type*/*

To test and debug your parser, you need to be able to display and examine the AST 1 by producing a flat program that corresponding to the tree. This process is called un-parsing.
The routine:

- *void print_ast-.node (FILE *f, AST *n) /*prints the AST node N */*

The routine:

- *int eva_ast_expr(File *f, AST *n) /*evaluates an AST (which should be a constant , integer expression) and returns its value*