Department of Computer Engineering and Informatics
"Translators"

Programming exercise:

The minimal++ programming language

Minimal++ is a small programming language built around the needs of the programming exercise of the course. Although its programming capabilities are small, this training language contains rich elements and the construction of its compiler is quite interesting, since it contains many commands used by other languages, as well as some original ones. Minimal++ supports functions and procedures, parameter passing by reference and value, recursive calls and other interesting structures. It also allows foliation in function declaration which few languages support (Pascal supports it, C does not).

On the other hand, minimal++ does not support basic programming tools such as the *for* structure, or data types such as real numbers and strings. These omissions have been made in order to simplify the compiler construction process, but a simplification that has only to do with the reduction of lines of code and not with the difficulty of construction or the educational value of the exercise.

Below is a description of the language:

Verbal units

The minimal++ alphabet consists of:

- the upper and lower case letters of the Latin alphabet ("A",...,..., "Z" and "a",...,..., "z"),
- the numeric digits ("0",...,..., "9"),
- the symbols for arithmetic operations ("+", "-", "*", "/"),
- the association operators "<", ">", "=", "<=", ">=", "<>",
- the assignment symbol ":=",
- the delimiters (";", ",", ":")
- as well as the grouping symbols ("(",")","[","]","{","}")

- and comment separation ("/*", "*/","//").

The symbols "[" and "]" are used in logical representations like the symbols "(" and ")" are used in numerical representations.

The reserved words are:

program	***declare***				
if	***then***	***else***			
while	~~***doublewhile***~~	~~***loop***~~	~~***exit***~~		
forcase	~~***incase***~~	***when***	***default***		
not	***and***	***or***			
function	***procedure***	***call***	***return***	***in***	***inout***
input	***print***				

These words cannot be used as variables. Language constants are integer constants consisting of an optional sign and a sequence of numeric digits.

Language identifiers are strings consisting of letters and digits, but starting with a letter. The compiler takes into account only the first thirty letters. White characters (tab, space, return) are ignored and can be used in any way without affecting the operation of the compiler, as long as they are not in bound words, identifiers, constants. The same applies to comments, which must be inside the **/*** and ***/** symbols or after the **//** symbol and up to the end of the line. It is forbidden to open comments twice before the first ones are closed. Empty comments are not supported.

Programme format

program *id*
{
    declarations
    subprograms
    sequence of statements
}

Types and declarations of variables

The only data type supported by minimal++ is integers. Integers must have values from -32767 to 32767. The declaration is done with the ***declare*** command. The identifier names follow without any other declaration, since we know they are integer variables and do not need to be on the same line. The variables are separated by commas. The end of the statement is identified by the Greek question mark. It is allowed to have more than one consecutive use of ***declare*****.**

Operators and expressions

The priority of the operators from largest to smallest is:

(1) Solitary logicians: "not"

(2) Multiples: "*", "/"

(3) Single prostheses: "+", "-"

(4) Binary prostheses: "+", "-"

(5) Relative "=", "<", ">", "<>", "<=", ">="

(6) Logical "and",

(7) Logical "or"

Structures of language

Assignment

Id := expression

Used to assign the value of a variable or constant, or an expression to a variable.

Decision if

if *(condition)*
statements[1]
*[****else***
statements][2]

The ***if*** decision command evaluates whether the *condition* is true, and if it is, then the *statements*[1] that follow it are executed. The ***else is*** not a required part of the command and is therefore enclosed in square brackets. The *statements*[2] following ***else*** are executed if the *condition* does not hold.

Repeat while

while **(***condition*)
statements

The **while** iteration command continuously repeats the *statements* as long as the *condition condition* holds. If the first time the *condition is* evaluated, the result of the evaluation is false, then the *statements are* never executed.

~~**Repeat loop**~~

~~***loop***~~
~~*statements*~~

~~The **loop** repeat command repeats the *statements* forever. Exit the loop only when the **exit** command is called~~

Repeat forcase

forcase
*(**when:** (condition)**:** statements*[1] *)**
default: *statements*[2]

The ***forcase*** iteration structure checks the *conditions* after ***when.*** Once one of them is found true, the following *statements*[1] are executed. Then the check jumps to the beginning of ***forcase***. If none of the ***when*** is true, then the check switches to ***default*** and the *statements*[2] are executed.

~~**Repeat incase**~~

~~***incase***~~
~~*(**when:** (condition**:** statements)**~~

~~The ***incase*** iteration structure checks the *conditions* that come after ***when***, examining them in order. For each of them for which the corresponding *condition* holds, the *statements* following the ":" symbol are executed. All *conditions* will be examined *and* all *statements* whose *conditions are* true will be executed. After all the ***when'***s have been examined, the control switches out of the ***incase*** structure if none of the *statements are* not~~

has been executed, or jumps to the beginning of the **incase** if at least one of the *statements* has been executed.

Repeat doublewhile

doublewhile *(condition)*
statements[1]
else
statements[2]

The first time the control enters the **doublewhile** loop, it is decided through the *condition* whether the execution will switch to *statements*[1] (*true*) or switch to *statements*[2] (*false*). It leaves *statements*[1] when the condition stops being *true*. From *statements*[2] it leaves when the condition stops being *false*. In both of these cases, control is passed outside the structure. That is, it is never possible in one execution of **doublewhile for** control to pass through both *statements*[1] and *statements*[2] .

Return function value

return *(expression)*

Used within functions to return the result of the function.

Data output

print *(expression)*

Displays the result of the *expression* valuation on the screen

Data input

input *(id)*

Prompts the user to enter a value from the keyboard

Calling procedure

call *function_name(actual_parameters)*

It calls for a procedure

Exit from loop loop

exit

Executes ***loop*** *exit*

Subprogrammes

minimal++ supports functions.

function *id* ***(****formal_pars****)***
{
declarations
subprograms
statements
}

Formal_pars is the list of formal parameters. Functions can nest inside each other and the scoping rules are like PASCAL. The return of a function's value is done with **return.**

The call of a function is made from the numeric representations as an operand. e.g.

D = a + f(**in** x)

where f is the function and x is a parameter passed by value.

The procedures are written as follows:

procedure *id* ***(****formal_pars****)***
{
declarations
subprograms
statements
}

The call of a procedure is done with **call**. e.g.

call f(**inout** x)

where *f* is the process and *x* is a parameter passed by reference.

Transmission of parameters

minimal++ supports two ways of transmitting parameters:

- with a fixed price. It is denoted by the verbal unit **in**. Changes to its value are not returned to the program that called the function.
- with reference. It is indicated by the verbal unit **inout**. Any change in its value is immediately passed to the program that called the function.

In a function call the actual parameters are drawn after the keywords

in and **inout**, depending on whether they are passed by value or reference.

End of

minimal++ files have the extension .min

The point:

Build in Python, a fully functional compiler of the minimal++ language. The compiler must produce the assembly language of the MIPS processor as the final language.

The laboratory exercise will be delivered in four phases: a)

verbal syntactic analysis

b) intermediate code generation

(c) semantic analysis and symbol table

(d) final code and report of correct operation check.