CISC / CMPE 458 Course Project Winter 2019
The MiniT Programming Language

J.R. Cordy - January 2019

The course project this year consists of implementing a compiler for a subset of the programming language Turing, called *MiniTuring (MiniT).* Turing is a modern Pascal successor language designed for teaching. In this project, we will implement MiniT by modifying and extending each of the four phases of the existing PT Pascal complier.

For each phase, I will give suggestions on how to make the necessary modifications to PT to complete the phase. These hints are intended to help you see at least one way that PT can be modified to achieve the desired result. However, you are not required to pay any attention to these suggestions, and you may choose to design modifications of your own. (Theorem: there are an infinite number of good ways to skin any given cat. Corollary: I don't know them all.) However, the MiniT language you implement must meet exactly the specification below.

Teams

The project will be undertaken in teams of four students. You are expected to choose your teammates on your own and submit team sign-up sheets to me by the beginning of class on Wednesday, January 23. The sign-up sheet must include the full name and student number of each team member, and must be signed by all four. Students who have not chosen teammates by January 23 will be assigned teammates at random.

The project forms an integral part of the course and you will be examined on material which can only be learned by actively taking part in each phase. You should therefore take care that every member of the team has an opportunity to contribute to every phase of the project. Assigning each phase's work to one person is not an appropriate way to split the work, because the phases require different amounts of work, ranging from 15% of the work for the parser phase to 35% of the work for the semantic phase, and because you will not all learn about all the phases that way. Assigning each language feature to one person is also not the best way to split the work, because some of the extensions will involve a lot of work in some phases and none at all in others, so you will not learn about all the phases that way.

The best way to manage the team is to hold team meetings to design overall solutions for each phase, estimate the work required to do the modifications, and then split the work as evenly as possible between team members. Team meetings are essential to the success of the project, and you should be sure to take time to hold them before each phase.

The MiniTuring Language

MiniTuring, or MiniT, is a modern modular language with features similar to languages such as Swift, Ruby, and Python. However, from our point of view, it is a modification and extension to PT Pascal. The main differences between PT and MiniT are Turing syntax for programs, statements and declarations, the addition of Turing **modules**, the replacement of the ***char*** data type with the Turing ***string*** type, the addition of Turing's otherwise clause to **case** statements, the addition of Turing's **elsif** to **if** statements, and the replacement of the PT **repeat** and **while** statements with the Turing general **loop** statement.

In the following,
  [item] means the item is optional, and
  { item } means zero or more repeated occurrences of the item.
  Keywords are shown in **bold face**, and predefined identifiers in ***bold italics***.
  Syntactic forms (nonterminals) not defined here are as defined in original PT Pascal.

1. Comments.

MiniT changes to Turing comments, which consist of % to end of line.
PT Pascal **{** ... **}** and **(*** ... ***)** comments are deleted from MiniT.

2. Programs.

A MiniT *program* is :

> **external** identifier { **,** identifier }
> { declaration }
> { statement }

Where *declaration* and *statement* are as described below.

Execution of the MiniT program consists of initializing the declarations in the order that they appear, and then executing the statements.

3. Declarations.

MiniT declarations are similar to PT, except for changes to modern Turing-like syntax.

A *declaration* is one of :

a. constantDeclaration
b. typeDeclaration
c. variableDeclaration
d. routine
e. module
f. **;**

A *constantDeclaration* is:

> **const** identifier :**=** constant

A *typeDeclaration* is:

> **type** identifier **:** type

A *variableDeclaration* is:

> **var** identifier { , identifier } **:** type

Where *constant* and *type* are as described in the PT Pascal syntax.

MiniT has no semicolons between or at the end of declarations, since they are redundant. However, semicolons are accepted as a null declaration (which does nothing, and thus can be used as a separator between other declarations).

4. Routines.

MiniT routines (procedures) are like PT routines except for the change to Turing-like syntax.

A *routine* is :

> **procedure** [*****] identifier [**(** [**var**] identifier **:** type_identifier
>   { , [**var**] identifier **:** type_identifier } **)**]
>   { declaration }
>   { statement }
> **end procedure**

The optional ***** before the routine name indicates an *exported* routine (one that can be called from outside of the module it is declared in). See "Modules" below.

5. Modules

One of the most powerful modern software engineering tools is the concept of information hiding. Modern programming languages include special syntactic forms for information hiding such as classes or modules. One of the weaknesses of PT Pascal is its lack of such a feature. MiniT solves this problem by adding Turing modules. Like “anonymous classes” in C++ and Java, MiniT modules are single-instance.

A *module* is :

```
module identifier
    { declaration }
    { statement }
end module
```

The purpose of the module is to hide the internal declarations, data structures and routines from the rest of the program. Outside of the module, the module’s internal data cannot be accessed, and only those routines declared as *exported* (using *****) may be called.

The purpose of the statements inside the module is to initialize the module's internal data. During execution, the statements of each module are executed once to initialize the module before commencing execution of the statements of the main program.

6. Statements.

MiniT changes the syntax of PT Pascal statements to Turing style, replacing the **begin**...**end** statement with explicit **end** markers on each statement form instead. MiniT replaces PT’s **repeat** and **while** statements with Turing’s general **loop**, adds Turing’s **elsif** to **if** statements, and Turing’s otherwise alternative (**label:**) to case statements. MiniT also adds the Turing short forms **+=**, **-=**, **++** and **—** for incrementing / decrementing simple variables.

A *statement* is one of the following :

```
a.  variable := expression
b.  variable_identifier [ += | −= ] expression
c.  variable_identifier [ ++ | — ]
d.  routine_identifier [ ( expression { , expression } ) ]
e.  if expression then
        { statement }
    { elsif expression then
        { statement } }
    [ else
        { statement } ]
    end if
f.  case expression of
        label expression { , expression } :
            { statement }
        { label expression { , expression } :
            { statement } }
        [ label :
            { statement } ]
    end case
g.  loop
        { statement }
        exit when expression
        { statement }
    end loop
h.  ;
```

Where *variable*, *expression* and *constant* are as defined in the PT Pascal syntax.

MiniT has no semicolons between or at the end of statements, since they are redundant. However, semicolons are accepted as a null statement (which does nothing, and thus can be used as a separator between other statements).

MiniT retains the PT Pascal input/output routines, but renames *read* and *write* to *get* and *put,* and *readln* and *writeln to getln* and *putln.*

7. Strings.

Text manipulation in standard Pascal is painful because of the necessity of shovelling characters around one-by-one. Modern languages like Turing solve this problem by providing a built-in varying-length string data type or library. MiniT adds the ***string*** data type to PT (replacing the ***char*** data type and the ***packed*** attribute, which are deleted from MiniT).

A ***string*** literal is any sequence of characters except the double quote enclosed between double quotes. Example :

```
"This is a string"
```

String variables take on the length of the last value that they were assigned. Example :

```
s := “hi"        #  length of s is now 2
s := "there"     #  length of s is now 5
```

There is an implementation-defined maximum length (1,023 characters) for string values. Varying length strings are implemented by storing the string value with an extra trailing character (ASCII NUL, the byte 0) marking the end of the string, as in Turing and C. Each ***string*** variable is actually allocated a fixed amount of storage (1,024 bytes) within which the string value is stored.

Strings can be input and output to/from text files and streams. On input, the string read consists of the characters from the next input character to the end of the input line.

Unlike arrays of ***char*** in PT Pascal, values of type ***string*** can be assigned and compared as a whole. Besides assignment and comparison, there are three new operations on strings: concatenation, substring and length.

Concatenation of strings is denoted by the + operator as in Turing. For example :

```
"hi " + "there"  =  "hi there"
```

It is an error to concatenate two strings if the sum of their lengths exceeds the implementation-defined maximum (detected at run time).

The MiniT substring operation is denoted by the **:** operator. The general form is :

```
expression @ expression .. expression
```

The first expression must be a ***string*** expression. The second and third expressions, which must be integer expressions, specify the (one origin) first character position and last character position of the substring respectively. Example :

```
"Hi there" @ 4 .. 6  =  "the"
```

The precedence of @ (including the **..** part) is the same as *****, **div**, **mod** and **and**.

The string length operator has the form :

```
? expression
```

Where the expression must be a ***string*** expression. Example :

```
? "Jim"  =  3
```

The precedence of **?** is the same as that of **not**.

8. Example MiniT Program

```
% Primes: determines the primes up to maxprimes using the sieve method
external output
const maxprimes := 100

% Prime flags
const prime := true
const notprime := false

module flags
    var flagvector: array [1 .. maxprimes] of boolean

    procedure * flagset (f: integer, tf: boolean)
        flagvector [f] := tf
    end procedure

    procedure * flagget (f: integer, var set: boolean)
        set := flagvector [f]
    end procedure

    % Everything begins as prime
    var i: integer
    i := 1
    loop
        exit when i > maxprimes
        flagvector [i] := prime
        i ++
    end loop
end module

% Main program
var multiple, factor : integer
var isprime: boolean

% Pick out multiples of each prime factor and set these to notprime
multiple := 2
factor := 2
loop
    exit when factor > maxprimes div 2

    % Set multiples of factor to notprime
    multiple := factor + factor
    loop
        exit when multiple > maxprimes
        flagset (multiple, notprime)
        multiple += factor
    end loop

    % Set factor = next prime
    factor ++
    loop
        flagget (factor, isprime)
        exit when (factor > maxprimes div 2) or isprime
        factor ++
    end loop
end loop

% Now report the results
put ("The primes up to ", maxprimes:1, " are:")
putln
factor := 2
loop
    exit when factor > maxprimes
    flagget (factor, isprime)
    if isprime then
        put (factor:4)
    end if
    factor ++
end loop
putln
```