About Final Project

Tsan-sheng Hsu

tshsu@iis.sinica.edu.tw

http://www.iis.sinica.edu.tw/~tshsu
Language definitions

- A static scoping language called $P$.
  - PASCAL-like;
  - lexical scoping;
  - block structure;
  - nested procedure with recursion;
  - case sensitive;
  - using reserved words;
    - All reserved words are upper cased.
  - use “;” as the statement terminator;
  - use “,” as the list separator.

- Requirements:
  - using LEX and YACC
  - generate C-- intermediate code
    - latest version: v 2.3
  - using C compiler to translate C-- code into machine object code
Template of a program (1/3)

- **header:**
  - PROGRAM name

- **constant definitions: optional**
  - CONST
    - single-name = (constant | a previously declared constant name); | ε
    - ...
    - ENDCONST

- **type definitions: optional**
  - TYPE
    - single-name = (default type | previously defined type name); | ε
    - ...
    - ENDTYPE

- **variable declarations: optional**
  - VAR
    - non-empty-list-of-names : type; | ε
    - ...
    - ENDVAR
Template of a program (2/3)

- **procedure/function definition:** can have 0, 1, 2, ... such definitions.
  - \(\text{PROCEDURE name parameters ;} \mid \text{FUNCTION name parameters : type;}\)
  - constant definitions: optional
  - type definitions: optional
  - variable definitions: optional
  - (procedure/function definition)*
  - block of statements

- **parameters:** \(\epsilon \mid () \mid \text{(lists)}\)
  - non-empty-list-of-names : type \| VAR non-empty-list-of-names : type
  - entries are separated by “;”
  - do not need “;” for the last entry
block of statements:

- example:
  - BEGIN
  - variable declarations: optional
  - (statement | block of statements)*
  - END

- variables declared inside a block are different, in term of scope, from the variables declared before a block, that is in the header area.
Example

PROGAM main

CONST %% can be empty or completely missing
   cons360 = 360; %% a legal name on the left, a legal constant on the right
   myfloat = 3.6;
ENDCONST

TYPE %% can be empty or completely missing
   mytype = ARRAY[1..10] OF INTEGER;
ENDTYPE

VAR %% can be empty or completely missing
   x : ARRAY[-3 .. 5] OF INTEGER;
   y : mytype;
ENDVAR

FUNCTION foo(x,y : INTEGER): INTEGER;
BEGIN
   foo := x * x - 3;
END

BEGIN
   BEGIN
      VAR
         w, x, z: INTEGER;
      ENDVAR
   x := foo(y[4]);
   WRITE(x);
   WRITESP();
   WRITE(y);
   WRITELN();
END
Constants and names

- **Format of constants:**
  - Allow leading zeros.
  - *In the decimal system, no binary or octal.*
  - *When constants cannot be represented by 32 bits, then they cause overflow errors.*
  - *REAL constant: integer.integer.*
  - *string constant: C style.*

- **names of variable, program, procedure or function:**
  - Legal C variable names;
  - Length of variable names: from 1 to 1024 characters;
  - Using ASCII encoding;
  - Names of program, procedure or function cannot be the same with variables or other names in the same scope;
Data types and variables

- **elementary types:**
  - INTEGER: 32-bit signed
  - REAL: 32-bit
  - INTEGER and REAL are not compatible types
  - New type defined is not elementary even when it is only renaming

- **aggregate types:**
  - 1-D array: ARRAY [ lower .. upper ] OF elementary type;
  - multi-D array: row major
    - ARRAY [ lower1 .. upper1, lower2 .. upper2, ...] OF elementary type;
  - lower and upper are integer constants and lower <= upper.
  - There is no space inside “..”, but there can be white spaces around “..”.
  - there can be spaces between ARRAY and [.

- **type equivalence: name equivalence**
  - check for incompatible types
I/O statements

- **READ**(non-empty-list-of-variables)
  - each variable must be of the type INTEGER or REAL;
  - data types of variables can be mixing;
  - variables are separated by “,”;

- **WRITE**(non-empty-list-of-variables/constants)
  - each variable/constant must be of the type INTEGER or REAL;
  - data types of variables/constants can be mixing;
  - variables are separated by “,”;
  - there is one blank in between two variables;

- **WRITESP()** — output a single space
  - white spaces are allowed around and in “()”

- **WRITELN()** — write a new line

- **WRITESTRING**(a C-string) — output a string in C format

- **Note:** in general, white spaces are allowed around “(“ and “)”.
Procedure and function (1/3)

- **Procedure**: one that does not return anything
  - Can only be called as
    - `procedure();`

- **Function**: one returns a value of the elementary type
  - The function name is a variable holding the returned value.
  - This variable has no r-value.
    - *If this name appears on the right hand side of “:=” then it is a function call.*

- **Procedure and function names:**
  - Their scope equals to the scope declaring them.
  - Procedure/function names are also used at the same time in the scope of their body.
    - *One cannot declare a variable named “www” inside a procedure/function named “www”.*
PROCEDURE p(x,y: INTEGER; VAR z: REAL);

VAR
    p : REAL;  % this is illegal
ENDVAR

FUNCTION foo(x:INTEGER): INTEGER;  % return value is INTEGER

VAR
    foo : REAL;  % this is illegal
ENDVAR

BEGIN
    foo := x * x;
END

BEGIN
    y := foo(x);
END
Procedure and function (3/3)

- **Parameters:**
  - call-by-value
    - name : type
  - call-by-reference
    - VAR name : type

- **Example:**

```pascal
PROCEDURE p(x,y: INTEGER; VAR z: REAL);
  FUNCTION foo(x:INTEGER): INTEGER; %% return value is INTEGER
  BEGIN
    foo := x * x;
  END
BEGIN
  y := foo(x);
END
```
Statements

- One line contains at most one statement.
  - comments: from %% to the rest of the line
  - “;” is statement terminator
  - a blank line is legal, but a line with only “;” is not legal;

- Any statement or declaration must be written in one line.
  - For example: header of a procedure

- Assignments and I/O statements.

- Procedure/function call statements.
  - p(100,200,w)
  - p()
  - The main program can recursively call itself.
  - Must check matched number and types of arguments.

- Return statement,
  - RETURN;
    - For a function, it automatically retrieve the current return value stored in the variable with the name equaling the function name.
Assignment and swapping

- **assignment**: `:=`
  - `variable := expression;`
  - `must be of the same type;`
  - `check for incompatible types;`

- **swap**: `<->`

  `a <-> b;` *%% swaps the content of two variables*
  - `swap two variables of identical types using name equivalence;`
  - `can be of any type;`
Operators

- precedence and associativity: same with the ANSI C language.
- arithmetic: $+, -, *, /, MOD$, where $MOD$ is remainder;
  - $MOD$ is only for INTEGERS;
- logical: $OR, AND, NOT, XOR$
- comparison: $>, <, =, <=, >=, <>$
  - Must between data of identical elementary type;
Expressions

- **arithmetic expression:**
  - operations on integers/reals
  - no auto-type conversion
  - detect incompatible types
  - can have “(” and “)”
  - Example:
    \[(x + y - 3) \ast 4 + 5\]

- **boolean expression: no short-circuited evaluation.**
  - Contents:
    - Basics: comparisons between equivalent-typed arithmetic expressions.
    - Apply logical operator on the above basics.
  - can have “(” and “)”
  - Example:
    \[(x > y) \ OR (z \geq 3.0)\]
  - The result of a boolean expression cannot be saved into any variable.
Conditional statements

- IF ... THEN ... ENDF;
- IF ... THEN ... ELSE ... ENDF;

IF boolean-expression
THEN
    statement / block of statements
ENDIF;

IF boolean-expression
THEN
    statement / block of statements
ELSE
    statement / block of statements
ENDIF;
Case statements

CASE expression OF
    constant_1 : statement/block of statment
    constant_2 : statement/block of statment
    ....
[optional]
    OTHERWISE : statement/block of statment
ENDIF;

▷ the types of constant, and expression must be equivalent;

▷ only allow integers;

▷ after one constant is matched, the statement terminates; no need to write “break” inside each case;

▷ OTHERWISE is for the “default” case and must be the last entry.
For loop

- **Two different formats**

  /* add 1 at a time */
  FOR var := int-expression-1 TO int-expression-2 DO
      statement / block of statements

  /* minus 1 at a time */
  FOR var := int-expression-1 DOWNTO int-expression-2 DO
      statement / block of statements

- var must be a declared integer variable
- if the loop is not executed, then the value of the loop variable stays unchanged
- int-expression-1 and int-expression-2 are evaluated only once when the loop is first entered
- if a loop is entered, then the value of var must be int-expression-2 after it is finished
Examples of for loops

i:=3;
FOR i:=1000 TO 10 DO
BEGIN
...
END
WRITE(i); % i is 3

FOR i:=1 TO 10 DO
BEGIN
...
END
WRITE(i); % i is 10

FOR i:=10 DOWNTO 2 DO
BEGIN
...
END
WRITE(i); % i is 2
While loop

- while loop

```plaintext
WHILE boolean-expression DO
  statement / block of statements
```
Scores

- Teams
  - Two persons per team
  - One person per team: project score \(*1.1\)

- Phases: in this order.
  - 1. (25%) simple expression language with two data types and block structure
  - 2. (30%) constant and typedef
  - 3. (35%) 1-D array and then multi-D array
  - 4. (50%) boolean expressions, conditional, branching and looping statements
  - 5. (70%) non-nested procedure/function with call-by-value parameters and recursive calls
  - 6. (80%) call-by-reference parameters
  - 7. (100%) nested procedure/function
Bonus

- Do these only when everything required is done.
  - record: + 10%
    - RECORD a, b: INTEGER; ENDRECORD;
    - A new elementary type
    - X.a to access a field
    - Need to allow array of records
    - Need to allow record having arrays as elements
  - pointer: +10%
    - ptr = ^INTEGER;
    - To access the content: *ptr
    - Need to allow array of pointers
    - Need to allow pointer of records
    - Do not allow pointer arithmetics
    - Can only swap, assign and de-reference.
  - run-time/compiler time checking: +10%
    - array bounds
    - divide by zero for both integer and float
Submitted packages (1/2)

- Format of your package: check out the TA’s web site.
- Your final project package must include
  - A make file that produces a compiler with the name “pcompiler”, compiles and runs all of your test programs.
    - “pcompiler file.p”
      generates an executable object file “p.out”
      to execute the compiled program: p.out
    - “pcompiler -a file.p”
      generates a C– code file named “file.out”
  - Subdirectories:
    - src
    - doc
    - tests
  - Common and fatal errors:
    - Using relative path name, not absolute path name.
    - Using standard packages, not add hoc ones.
    - Setting up the right environments.
    - Specify contact information in case of emergency.
Submitted packages (1/2)

- **Documentation (in PDF, PS, TXT or HTML format):**
  - Language reference manual: language.xxx
  - List of features implemented and their corresponding test programs: features.xxx
  - Implementation manual: internal.xxx contains the implementation details.
  - Other helpful documents: otherX.xxx
  - Can merge everything into one document with clearly marked sections of the above. Call this file document.xxx

- **A collection of test programs, inputs and anticipated outputs.**
  - programX.p: program.
  - inputX_Y: input test data.
  - outputX_Y: output data.
  - readmeX: documentation for programX, contains the purpose of having test programX.
  - Example: program1.p, input1_1, input1_2, output1_1, output1_2 and readme1.
Grading

- **Correctness (50%)**:  
  - 35%: produce right codes on correct programs in a reasonable amount of time.  
  - 15%: detect and report errors on incorrect programs.

- **Documentation and Testing (30%)**:  
  - 20%: manuals.  
  - 10%: designs of your own set of test programs.

- **Elegance (20%)**:  
  - 5%: algorithmic issues.  
  - 5%: exact, helpful and nice error reporting.  
  - 5%: coding.  
  - 5%: optimization and other helpful features.