CMPE 152: Compiler Design

September 19 Lab

Department of Computer Engineering
San Jose State University

Fall 2017
Instructor: Ron Mak

www.cs.sjsu.edu/~mak

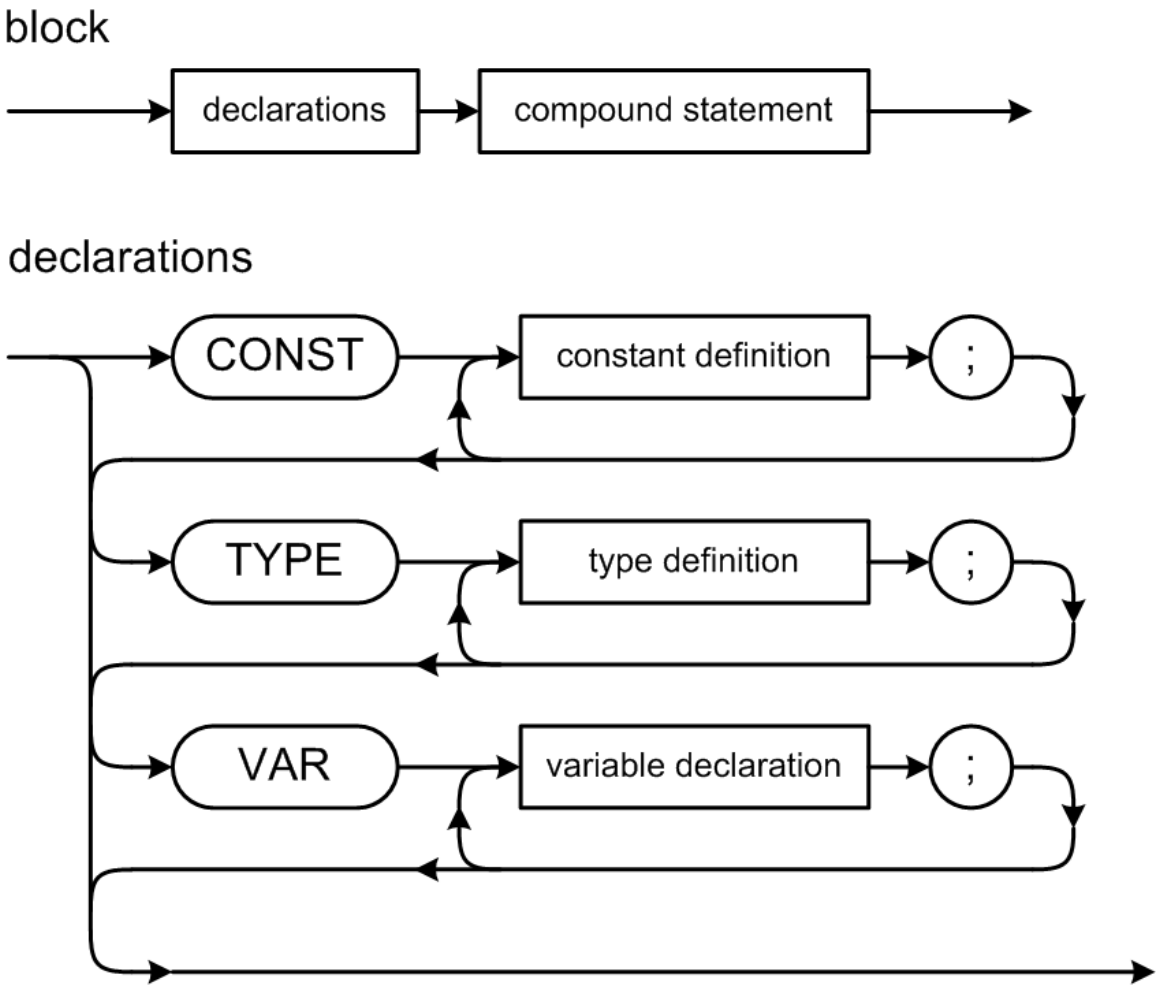
Pascal Declarations

- Classic Pascal declarations consist of 5 parts, each optional, but always in this order:

  1. Label declarations
  2. Constant definitions
  3. Type definitions
  4. Variable declarations
  5. Procedure and function declarations

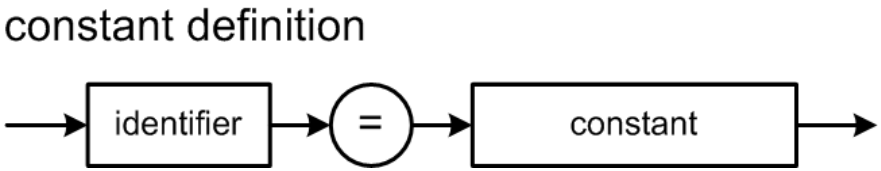
- We will examine 2, 3, and 4 next.
  - We'll do procedures and functions in a couple of weeks.

Pascal Declarations



- The **CONST**, **TYPE**, and **VAR** parts are optional, but they must come in this order.
- Note that constants and types are **defined**, but variables are **declared**.
- Collectively, you refer to all of them as declarations.

Pascal Constant Definitions



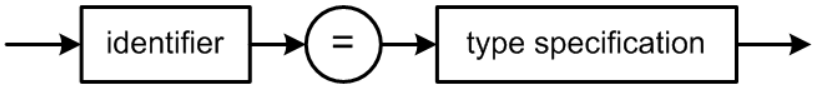
- Example constant definition part:

```
CONST
    factor = 8;
    epsilon = 1.0e-6;
    ch = 'x';
    limit = -epsilon;
    message = 'Press the OK button to confirm your selection.';
```

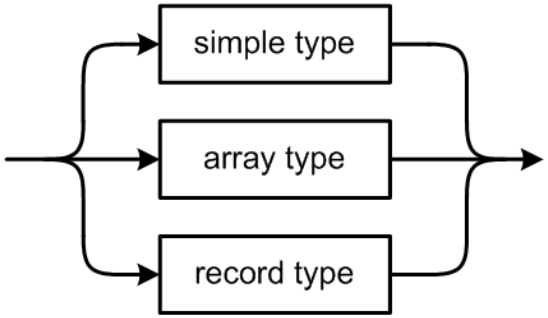
- Classic Pascal only allows a constant value after the = sign.
  - No constant expressions.

Pascal Type Definitions

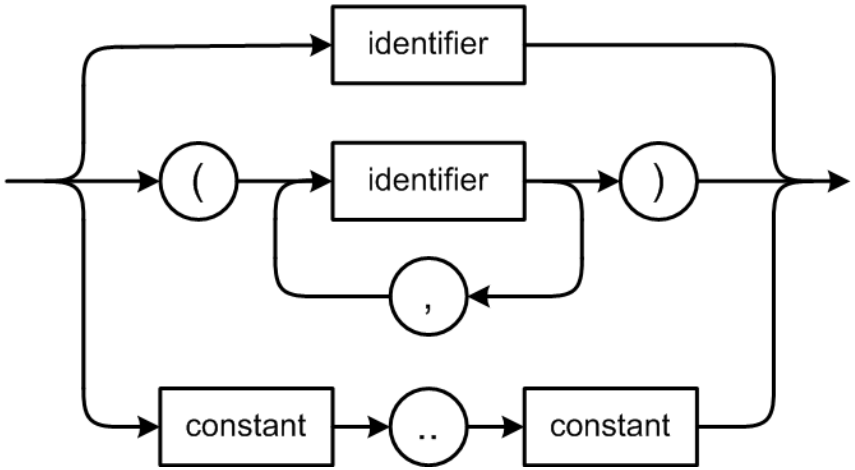
type definition



type specification



simple type



- A Pascal simple type can be:
  - scalar (**`integer`**, **`real`**, **`boolean`**, **`char`**)
  - enumeration
  - subrange

Not reserved words!

Pascal Simple Type Definitions

- Examples of subrange and enumeration type definitions:

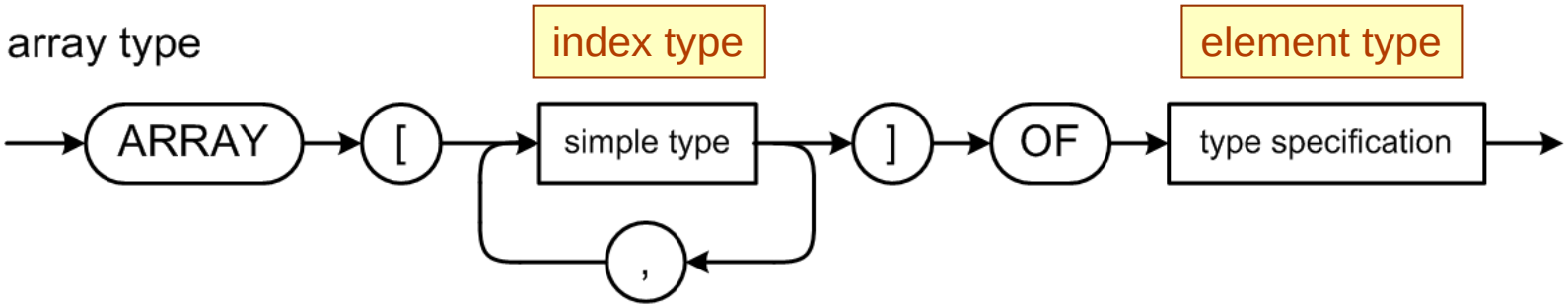
```
CONST
    factor = 8;

TYPE
    range1 = 0..factor; {subrange of integer (factor is constant)}
    range2 = 'a'..'q';  {subrange of char}
    range3 = range1;    {type identifier}

    grades  = (A, B, C, D, F);  {enumeration}
    passing = A..D;             {subrange of enumeration}

    week    = (monday, tuesday, wednesday, thursday,
               friday, saturday, sunday);
    weekday = monday..friday;
    weekend = saturday..sunday;
```

Pascal Array Type Definitions



- An array type specification has an index type and an element type.
- The index type must be a simple type (subrange or enumeration).
- The element type can be any type.
  - Including another array type (multidimensional arrays).

Pascal Array Type Definitions

- Examples of array definitions.

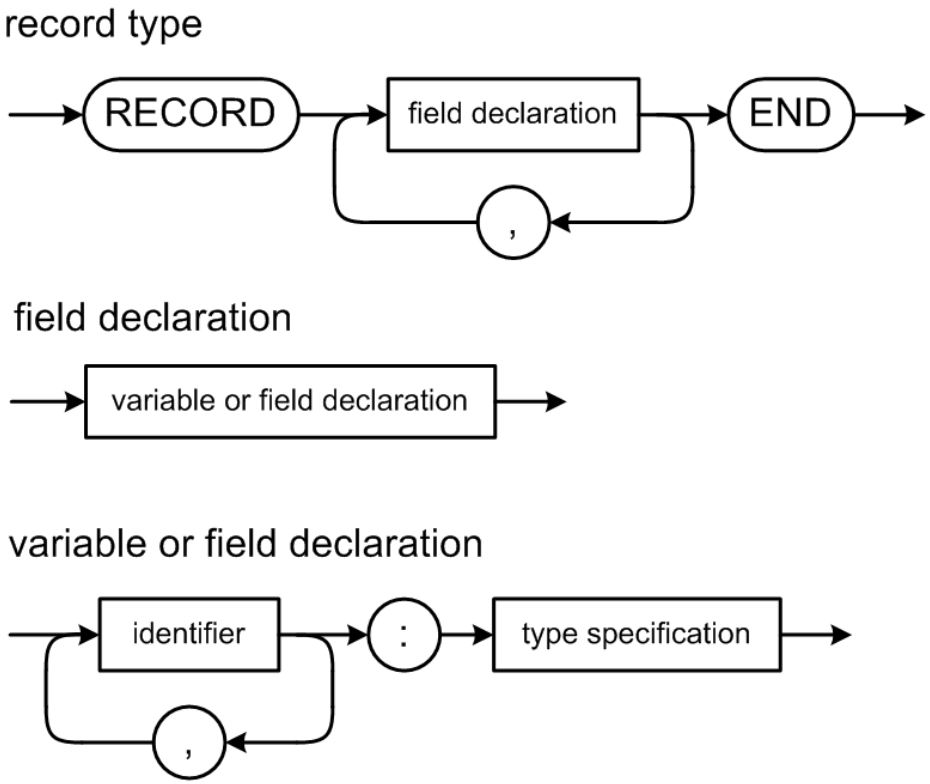
```
TYPE
    ar1 = ARRAY [grades] OF integer;
    ar2 = ARRAY [(alpha, beta, gamma)] OF range2;
    ar3 = ARRAY [weekday] OF ar2;
    ar4 = ARRAY [range3] OF (foo, bar, baz);
    ar5 = ARRAY [range1] OF ARRAY [range2] OF ARRAY[c..e] OF enum2;
    ar6 = ARRAY [range1, range2, c..e] OF enum2;
```

Type definitions **ar5** and **ar6** above are equivalent ways to define a multidimensional array.

- A Pascal string type is an array of characters.
  - The index type must be an integer subrange with a lower limit of 1.

```
str = ARRAY [1..10] OF char;
```

Pascal Record Type Definitions



- A record field can be any type.
  - Including another record type (nested records).

Pascal Record Type Definitions

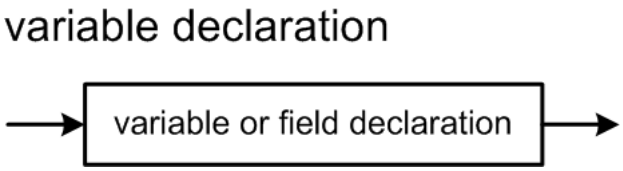
- Examples of record definitions:

```
TYPE
    rec1 = RECORD
               i : integer;
               r : real;
               b1, b2 : boolean;
               c : char
           END;

    rec2 = RECORD
               ten : integer;
               r : rec1;
               a1, a2, a3 : ARRAY [range3] OF range2;
           END;
```

Pascal Variable Declarations

- Variable declarations are syntactically similar to record field declarations:



- Examples:

```
VAR
    var1 : integer;
    var2, var3 : range2;
    var4 : ar2
    var5 : rec1;
    direction : (north, south, east, west);
```

- Types can be named or unnamed.

Declarations and the Symbol Table

- Identifiers from Pascal declarations that we will enter into a symbol table, names of:
  - constants
  - types
  - enumeration values
  - record fields
  - variables
- Information from parsing type specifications:
  - simple types
  - array types
  - record types

Scope and the Symbol Table Stack

- Scope refers to the part of the source program where certain identifiers can be used.
- Everywhere in the program where the definitions of those identifiers are in effect.
- Closely related to nesting levels and the symbol table stack.

Scope and the Symbol Table Stack, *cont'd*

- Global scope
  - Nesting level 0:
    At the bottom of the symbol table stack.
  - All predefined global identifiers,
    such as **`integer`**, **`real`**, **`boolean`**, **`char`**.
- Program scope
  - Nesting level 1:
    One up from the bottom of the stack.
  - All identifiers defined at the "top level"
    of a program (not in a procedure or function).

Scope and the Symbol Table Stack, *cont'd*

- Record definitions, procedures, and functions each has a scope.
- Scopes in a Pascal program are nested.
  - An identifier can be redefined within a nested scope.
  - Within the nested scope, the definition in the nested scope overrides the definition in an outer scope.
- Each scope must have its own symbol table.

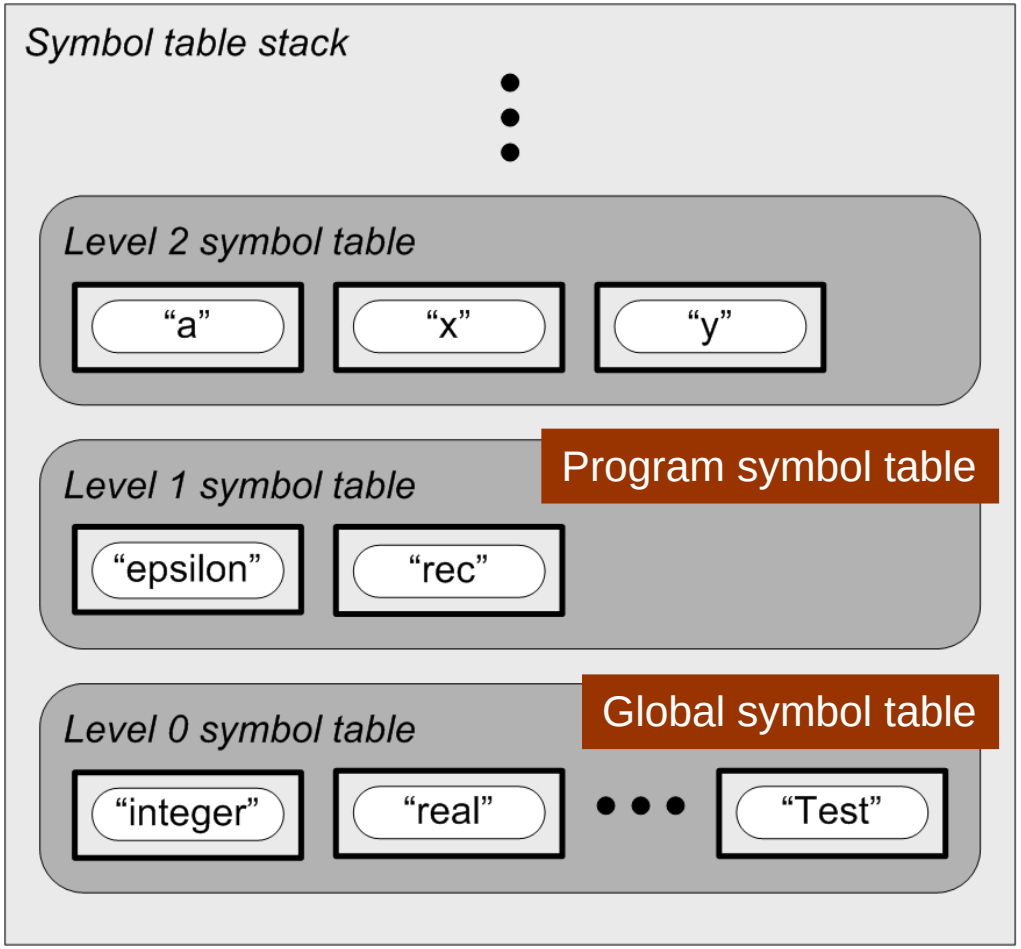
Scope and the Symbol Table Stack, *cont'd*

- As the parser parses a program from top to bottom, it enters and exits nested scopes.
- Whenever the parser enters a scope, it must push that scope's symbol table onto the symbol table stack.
- Whenever the parser exits a scope, it must pop that scope's symbol table off the stack.

Scope and the Symbol Table Stack, *cont'd*

- Scope example:

```
PROGRAM Test;
CONST
    epsilon = 1.0e-6;
TYPE
    rec = RECORD
              a : real;
              x, y : integer;
          END;
●●●
```



Note that the program name **`Test`** is defined in the global scope at level 0.

New Methods for Class **SymTabStackImpl**

```
public SymTab push()
{
    SymTab symTab = SymTabFactory.createSymTab(++currentNestingLevel);
    add(symTab);
    return symTab;
}

public SymTab push(SymTab symTab)
{
    ++currentNestingLevel;
    add(symTab);
    return symTab;
}

public SymTab pop()
{
    SymTab symTab = get(currentNestingLevel);
    remove(currentNestingLevel--);
    return symTab;
}
```

Push a new symbol table onto the stack.

Push an existing symbol table onto the stack.

Pop a symbol table off the stack.

Recall that we implemented **SymTabStackImpl** as an **ArrayList<SymTab>**.

Class **SymTabStackImpl**

```
public SymTabEntry lookupLocal(String name)
{
    return get(currentNestingLevel).lookup(name);
}

public SymTabEntry lookup(String name)
{
    SymTabEntry foundEntry = null;

    for (int i = currentNestingLevel; (i >= 0) && (foundEntry == null); --i)
    {
        foundEntry = get(i).lookup(name);
    }

    return foundEntry;
}
```

- Method **`lookup()`** now searches the current symbol table and the symbol tables lower in the stack.
  - It searches in the current scope and then outward in the enclosing scopes.