CS 153: Concepts of Compiler Design

September 26 Class Meeting

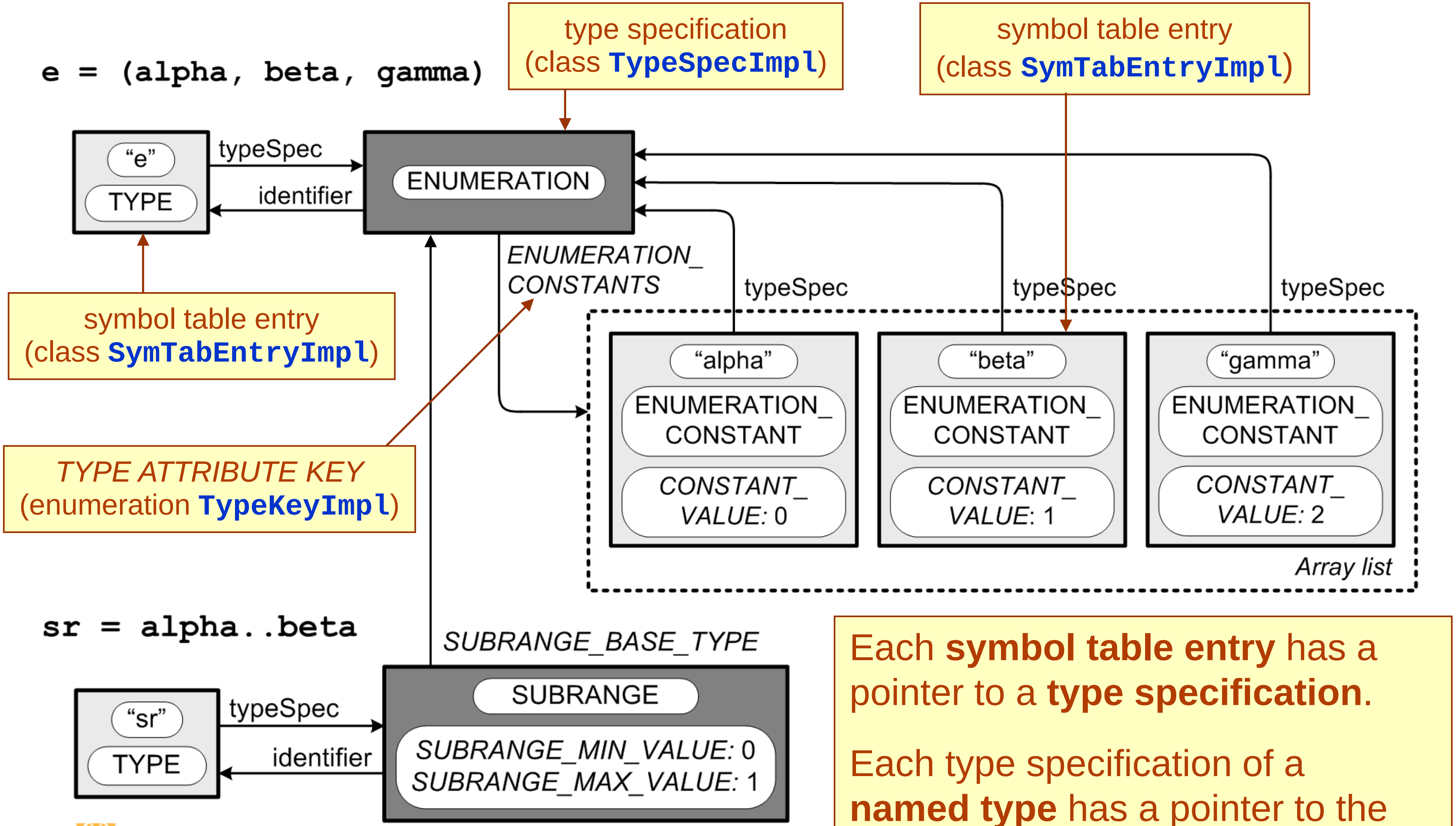
Department of Computer Science
San Jose State University

Fall 2017
Instructor: Ron Mak
www.cs.sjsu.edu/~mak

Type Definition Structures

`e = (alpha, beta, gamma)`

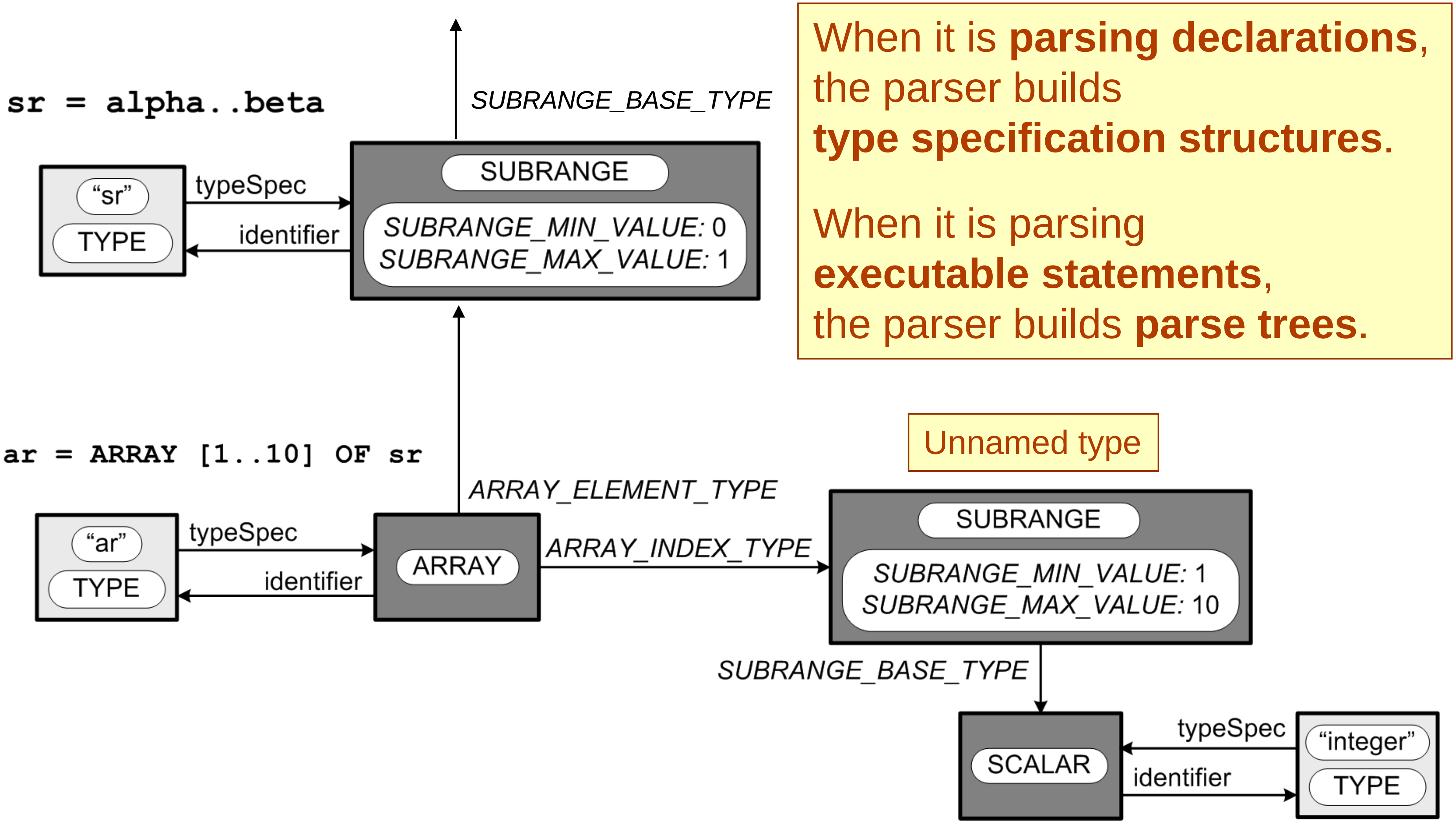
`sr = alpha..beta`



Each **symbol table entry** has a pointer to a **type specification**.

Each type specification of a **named type** has a pointer to the type identifier's symbol table entry.

Type Definition Structures, *cont'd*



```
sr = alpha..beta
```

```
ar = ARRAY [1..10] OF sr
```

When it is **parsing declarations**, the parser builds **type specification structures**.

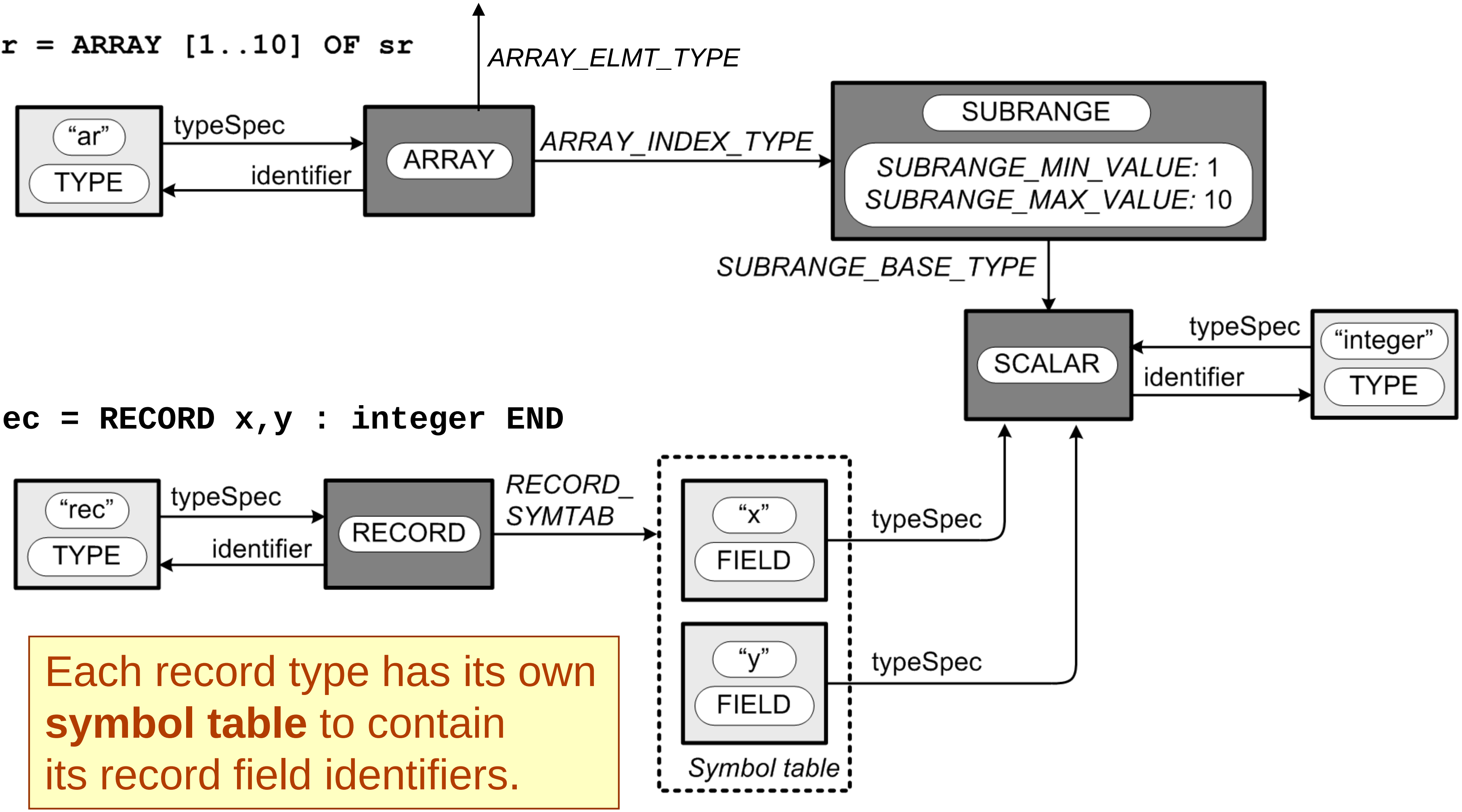
When it is parsing **executable statements**, the parser builds **parse trees**.

Unnamed type

Type Definition Structures, *cont'd*

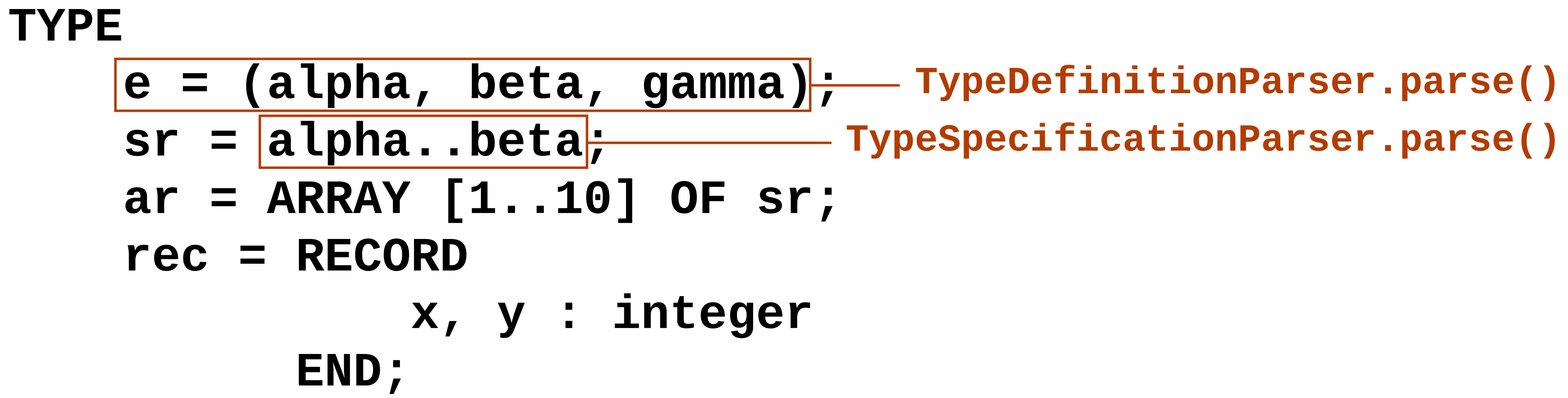
```
ar = ARRAY [1..10] OF sr
```

```
rec = RECORD x,y : integer END
```



Each record type has its own **symbol table** to contain its record field identifiers.

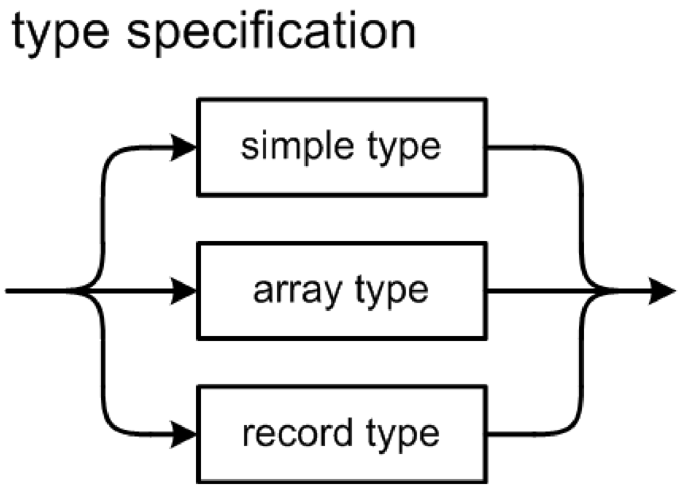
TypeDefinitionsParser.parse()



```
TYPE
    e = (alpha, beta, gamma);
    sr = alpha..beta;
    ar = ARRAY [1..10] OF sr;
    rec = RECORD
              x, y : integer
          END;
```

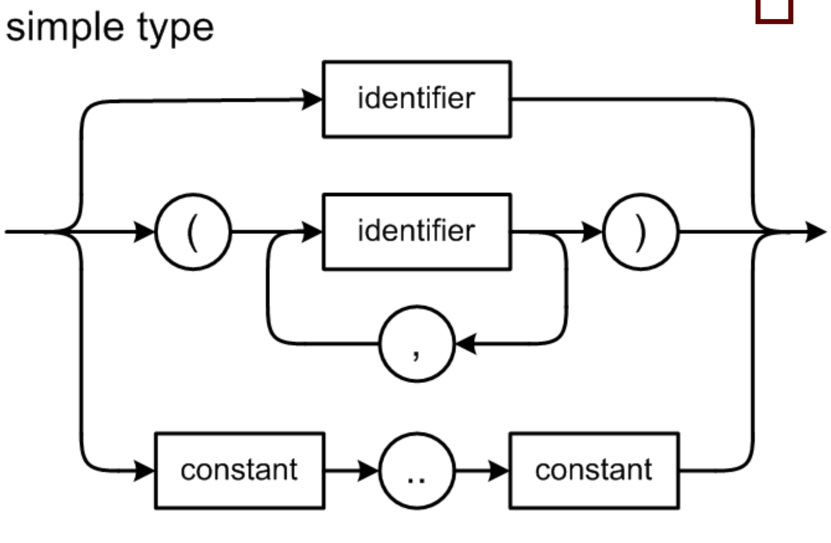
- Loop to parse each type definition.
  - Parse the type identifier and the **=** sign.
  - Call the **parse()** method of **TypeSpecificationParser**.
    - Parse the type specification and return a **TypeSpec** object.
    - Cross-link the **SymTabEntry** object of the type identifier with the **TypeSpec** object.

TypeSpecificationParser.parse()



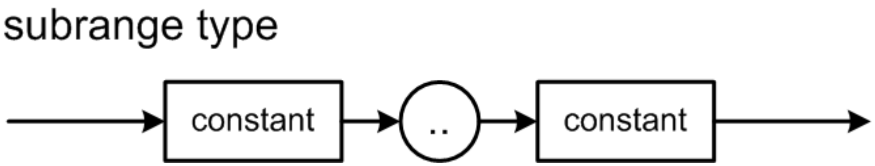
- Parse an array type.
  - If there is an **ARRAY** reserved word.
  - Call the **parse()** method of **ArrayTypeParser**
- Parse a record type.
  - If there is a **RECORD** reserved word.
  - Call the **parse()** method of **RecordTypeParser**
- Parse a simple type.
  - In all other cases.
  - Call the **parse()** method of **SimpleTypeParser**

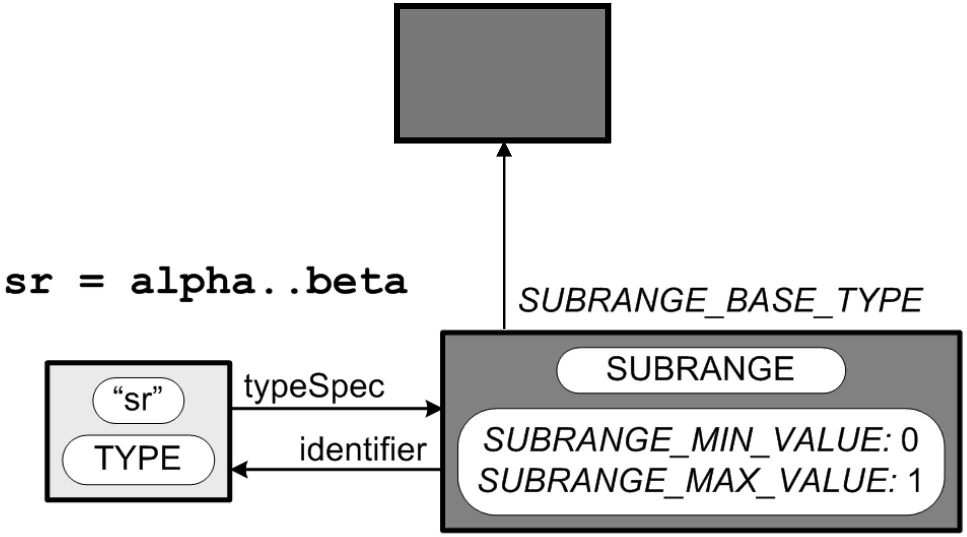
SimpleTypeParser.parse()



- Method **parse()** parses:
  - A previously-defined type identifier.
    - Including **integer**, **real**, etc.
  - An enumeration type specification.
    - Call the **parse()** method of **EnumerationTypeParser**.
  - A subrange type specification.
    - Call the **parse()** method of **SubrangeTypeParser**.

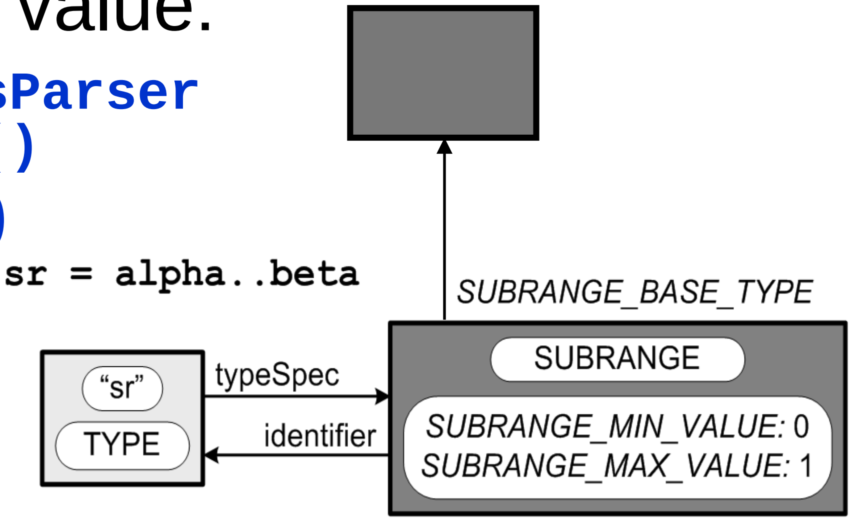
Pascal Subrange Type





SubrangeTypeParser.parse()

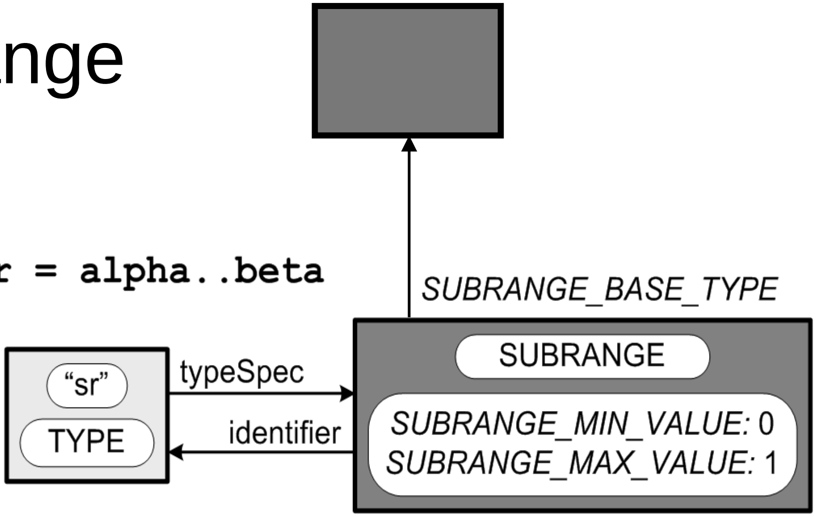
- Call **TypeFactory.createType(SUBRANGE)** to create a new subrange type specification.
- Parse the minimum constant value.
  - Call **ConstantDefinitionsParser.parseConstant()**
- Get and check the data type of the minimum constant value:
  - Call **ConstantDefinitionsParser .getConstantType()**
  - Call **checkValueType()**
    - The type must be integer, character, or an enumeration.
- Consume the **..** token.



SubrangeTypeParser.parse()

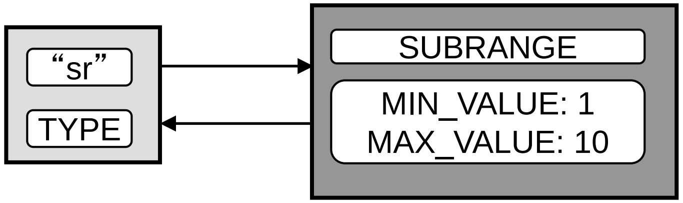
- Parse the maximum constant value.
- Check that both minimum and maximum values have the same data type.
- Check that the minimum value <= maximum value.
- Set attributes of the subrange type specification.
  - **SUBRANGE_BASE_TYPE**
  - **SUBRANGE_MIN_VALUE**
  - **SUBRANGE_MAX_VALUE**

```
sr = alpha..beta
```



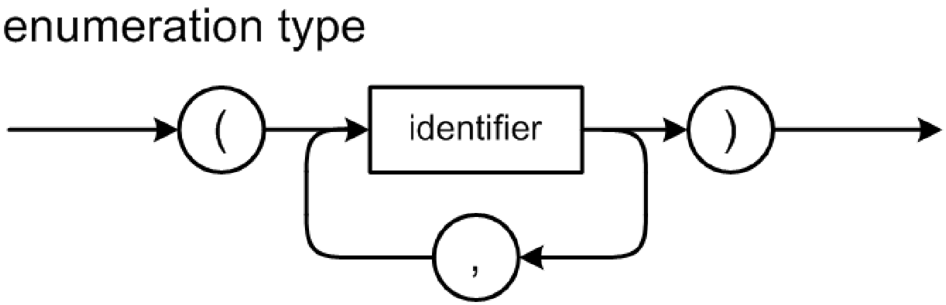
Parsing a Subrange Type

```
TYPE
    sr = 1..10;
    enum = (alpha, beta, gamma);
    ar = ARRAY [sr, enum] OF integer;
    rec = RECORD
              x, y : real
          END;
```

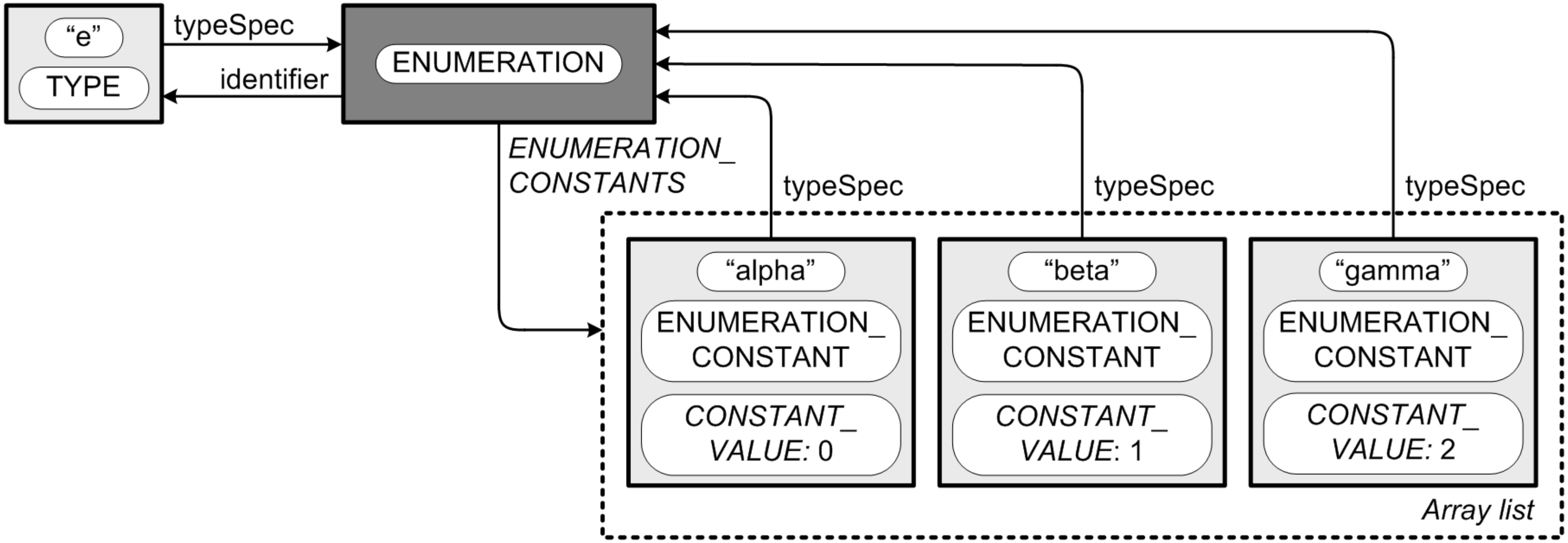


- **Pascal**
  - ➔ **PascalParserTD.parse()**
  - ➔ **BlockParser.parse()**
  - ➔ **DeclarationsParser.parse()**
  - ➔ **TypeDefinitionsParser.parse()**
    - ➔ **TypeSpecificationParser.parse()**
    - ➔ **SimpleTypeParser.parse()**
    - ➔ **SubrangeTypeParser.parse()**

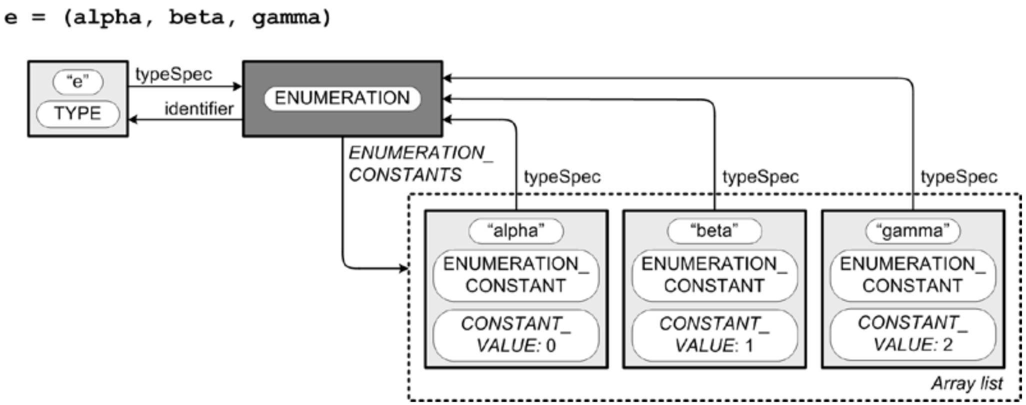
Pascal Enumeration Type



```
e = (alpha, beta, gamma)
```



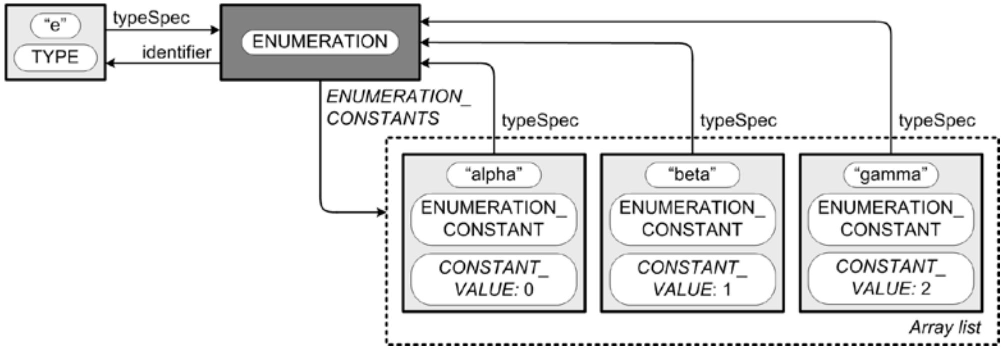
EnumerationTypeParser.parse()



- Call **`TypeFactory.createType(ENUMERATION)`** to create a new enumeration type specification.

`EnumerationTypeParser.parse()` *cont’d*

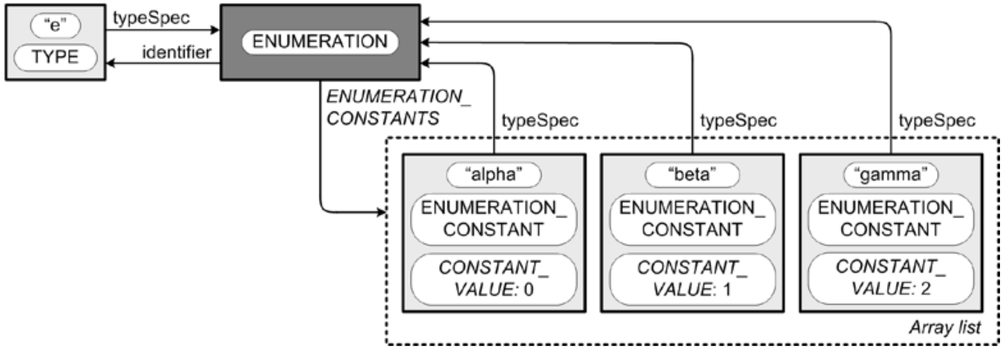
```
e = (alpha, beta, gamma)
```



- Loop to parse each enumeration identifier.
  - Call `parseEnumerationIdentifier()`
    - Set the definition of the identifier to `ENUMERATION_CONSTANT`.
    - Set the `typeSpec` field of the identifier to the enumeration type specification.
    - Set the `CONSTANT_VALUE` of the identifier to the next integer value (starting with 0).
  - Build an `ArrayList<SymTabEntry>` of symbol table entries for the enumeration identifiers.

EnumerationTypeParser.parse() *cont'd*

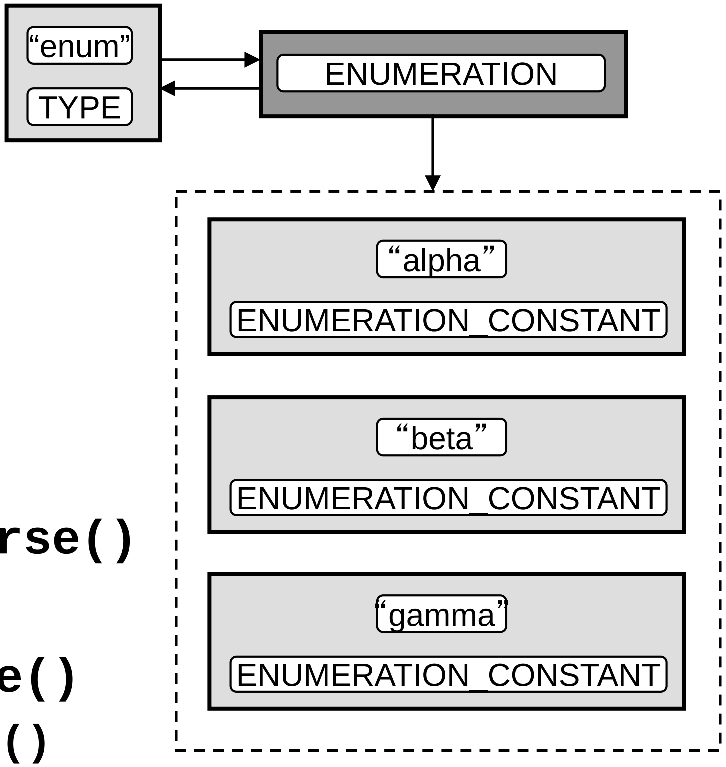
```
e = (alpha, beta, gamma)
```



- □ Set the **ENUMERATION_CONSTANTS** attribute of the enumeration type specification to the array list.

Parsing an Enumeration Type

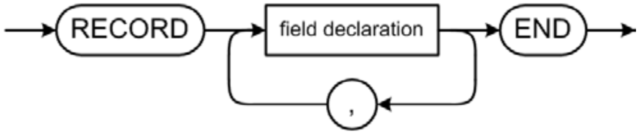
```
TYPE
    sr = 1..10;
    enum = (alpha, beta, gamma);
    ar = ARRAY [sr, enum] OF integer;
    rec = RECORD
              x, y : real
          END;
```



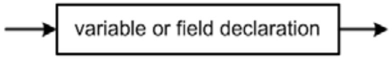
- TypeDefinitionsParser.parse()
  - ➔ TypeSpecificationParser.parse()
  - ➔ SimpleTypeParser.parse()
  - ➔ EnumerationTypeParser.parse()
    - ➔ parseEnumerationIdentifier()

Pascal Record Type

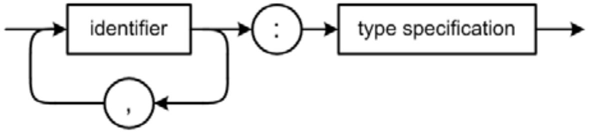
record type



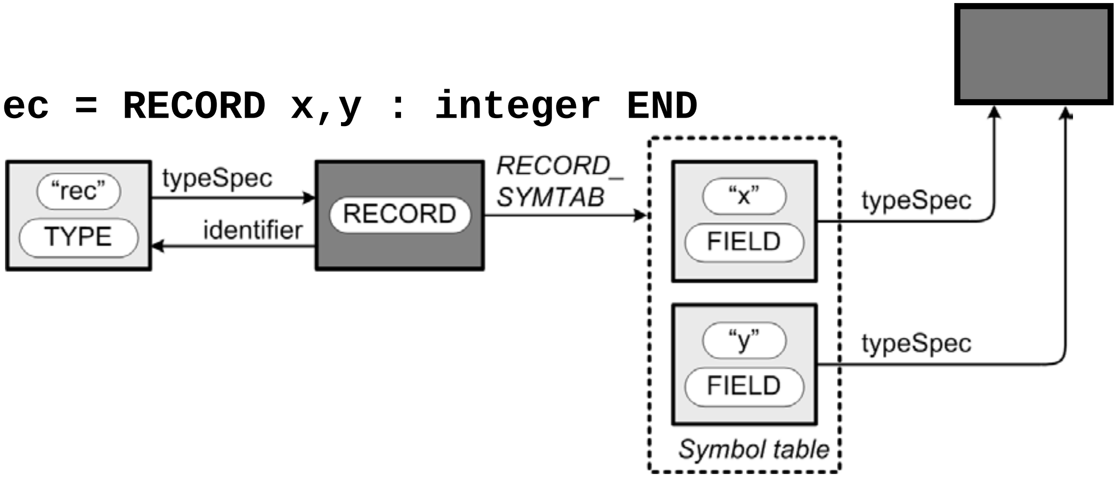
field declaration



variable or field declaration



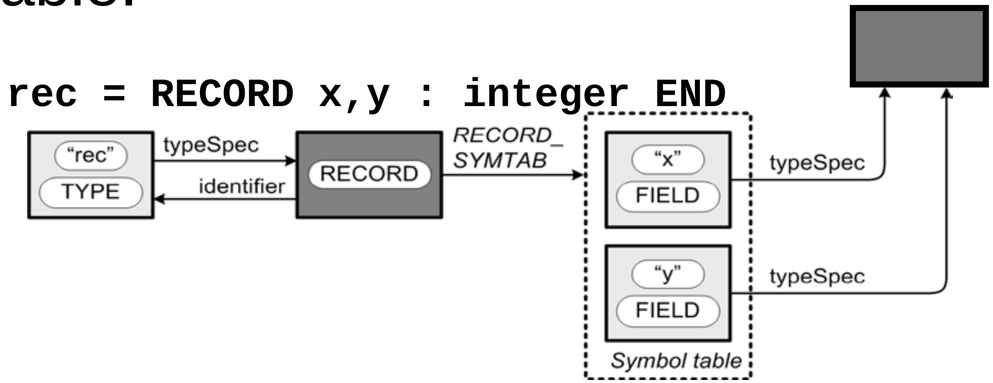
```
rec = RECORD x,y : integer END
```



`RecordTypeParser.parse()`

- Call **`TypeFactory.createType(RECORD)`** to create a new record type specification.
- Create and push a new symbol table onto the symbol table stack.
  - Set the **`RECORD_SYMTAB`** attribute of the record type specification to the new symbol table.

```
rec = RECORD x,y : integer END
```



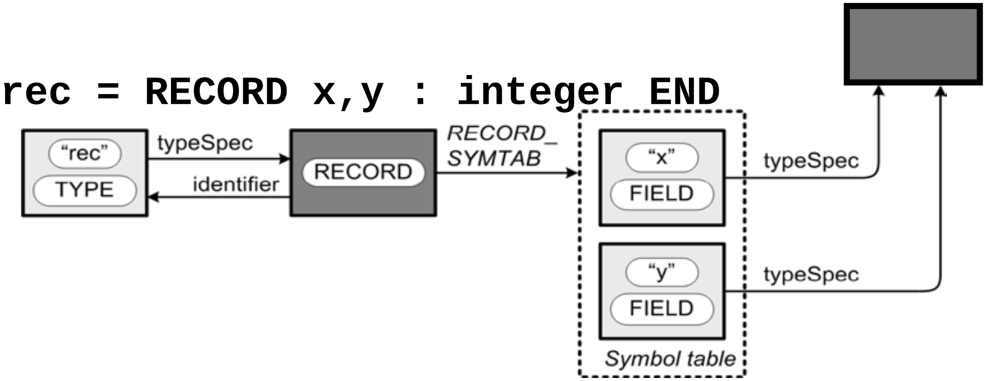
RecordTypeParser.parse() *cont'd*

- Call **VariableDeclarationsParser.parse()** to parse the field declarations.
  - Set each field's definition to **FIELD**.
  - Enter each field into the current symbol table (the one just pushed onto the top of the stack).

- Pop the record type's symbol table off the symbol table stack.

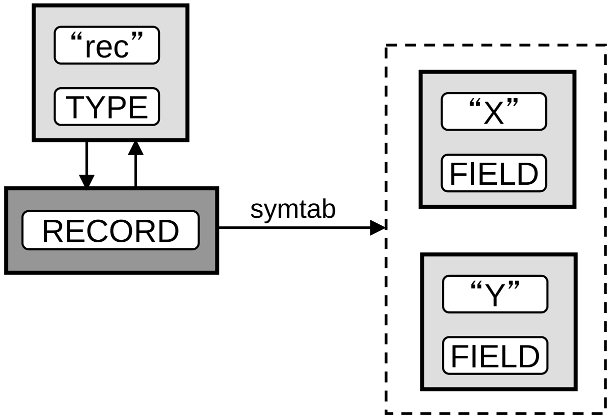
After the record type's symbol table has been popped off the symbol table stack, it's still referenced by the **RECORD_SYMTAB** attribute.

rec = RECORD x,y : integer END



Parsing a Record Type

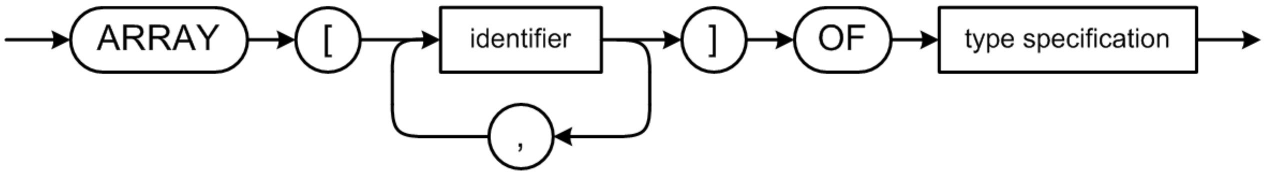
```
TYPE
    sr = 1..10;
    enum = (alpha, beta, gamma);
    ar = ARRAY [sr, enum] OF integer;
    rec = RECORD
                x, y : real
          END;
```



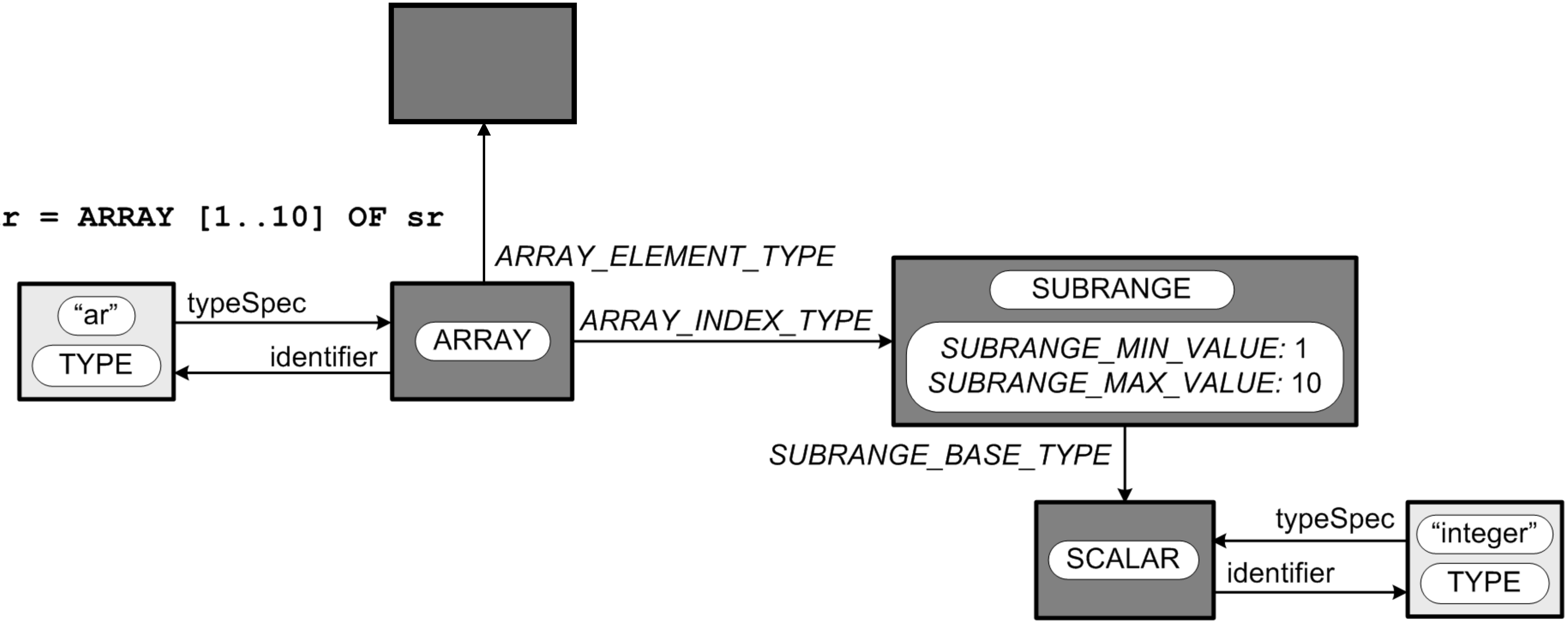
- **TypeDefinitionsParser.parse()**
  - → **TypeSpecificationParser.parse()**
  - → **RecordTypeParser.parse()**
    - → **VariableDeclarationsParser.parse()**

Pascal Array Type

array type



```
ar = ARRAY [1..10] OF sr
```



Pascal Multidimensional Array

- These definitions are all equivalent:

```
dim3 = ARRAY [1..3, 'a'..'z', boolean] OF real;

dim3 = ARRAY [1..3] OF ARRAY ['a'..'z']
                           OF ARRAY [boolean] OF real;

dim3 = ARRAY [1..3, 'a'..'z'] OF ARRAY [boolean] OF real;

dim3 = ARRAY [1..3] OF ARRAY ['a'..'z, boolean] OF real;
```

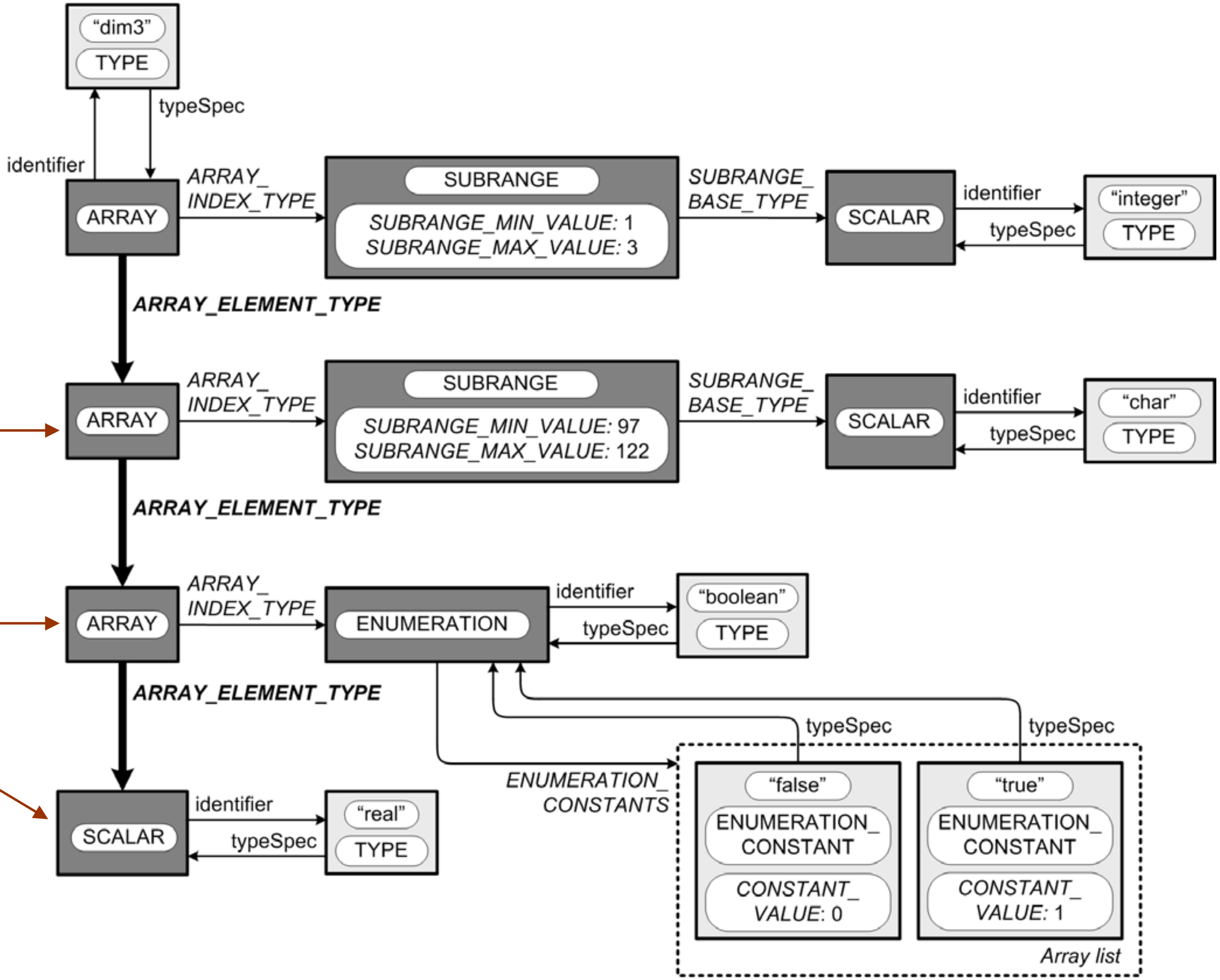
- Therefore, they must all generate the same type specification structure.

Pascal Multidimensional Array

```
dim3 = ARRAY [1..3, 'a'..'z', boolean] OF real;
dim3 = ARRAY [1..3] OF ARRAY ['a'..'z'] OF ARRAY [boolean] OF real;
```

The ***ARRAY_ ELEMENT_TYPE*** references form the **backbone** of this structure (a linked list).

The first two element types are **arrays**, and the third element type is **real** scalar.



`ArrayTypeParser.parse()`

- Call `TypeFactory.createType(ARRAY)` to create a new array type specification.

`ArrayTypeParser.parse()`

- Call `parseIndexTypeList()` to parse the list of index types.
  - Set local variable `elementType` to the new array type specification.
  - Loop to call `parseIndexType()` to parse each index type.
    - Call `simpleTypeParser.parse()` to parse the index type.
    - Set the attributes for a subrange or enumeration index type specification.
    - Set the `ARRAY_ELEMENT_COUNT` attribute for the current array type spec.

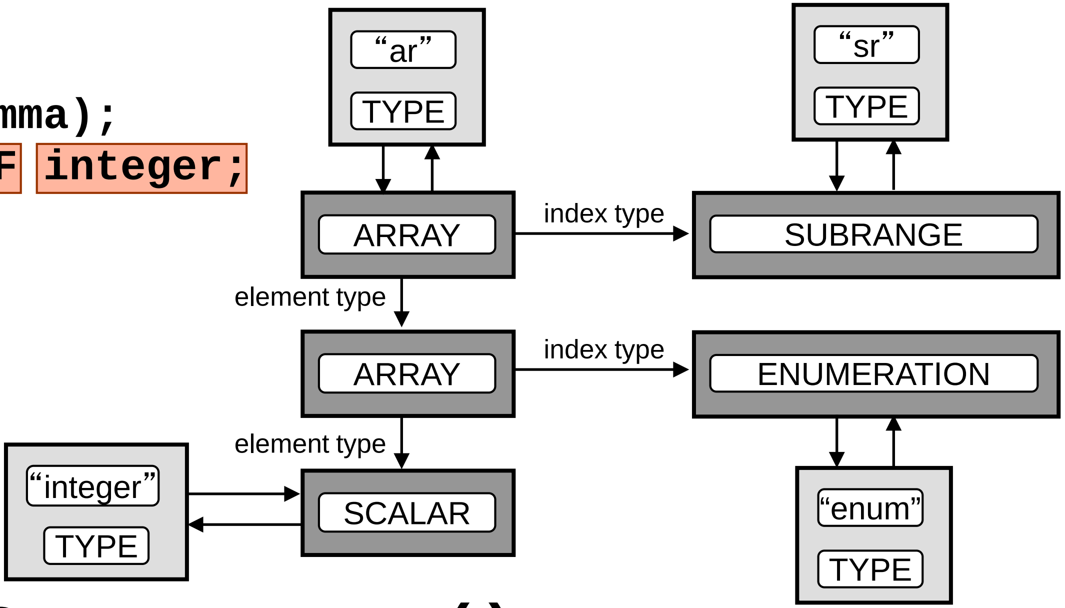
ArrayTypeParser.parse()

- Call **parseIndexTypeList()** to parse the list of index types *(cont'd)*.
  - For each index type in the list *after the first*:
    - Call **TypeFactory.createType(ARRAY)** to create the next **elementType** value.
    - Set the **ARRAY_ELEMENT_TYPE** attribute of the previous **elementType** value to link to the new **elementType** value.
- Call **parseElementType()** to parse the final element type.
  - Link the previous element type to the final element type.

These **elementType** references create the **backbone**.

Parsing an Array Type

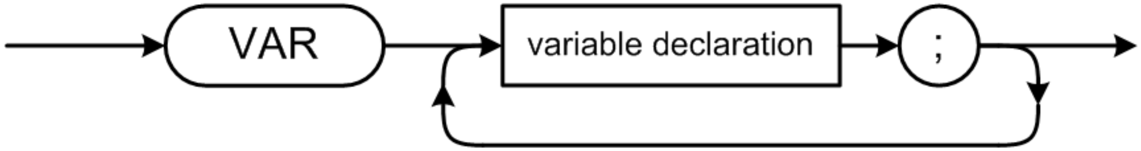
```
TYPE
    sr = 1..10;
    enum = (alpha, beta, gamma);
    ar = ARRAY [sr, enum] OF integer;
    rec = RECORD
              x, y : real
          END;
```



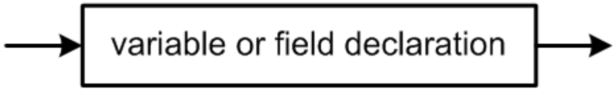
- `TypeDefinitionsParser.parse()`
  - ➔ `TypeSpecificationParser.parse()`
  - ➔ `ArrayTypeParser.parse()`
    - ➔ `parseIndexTypeList()`
    - ➔ `parseIndexType()`
    - ➔ `parseElementType()`

Pascal Variable Declarations

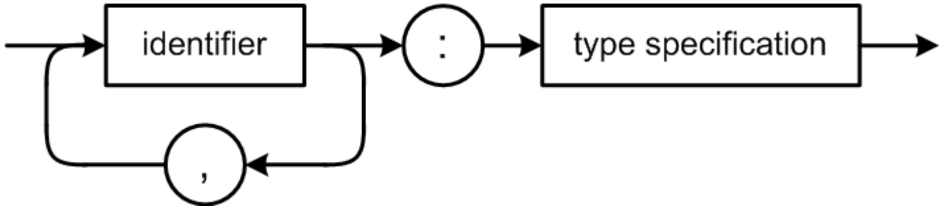
variable declarations



variable declaration

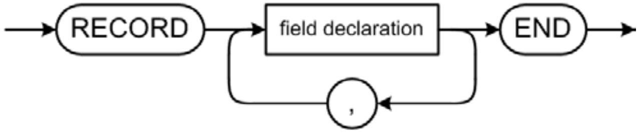


variable or field declaration

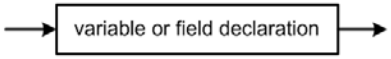


Compare to:

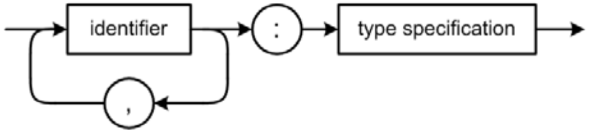
record type



field declaration

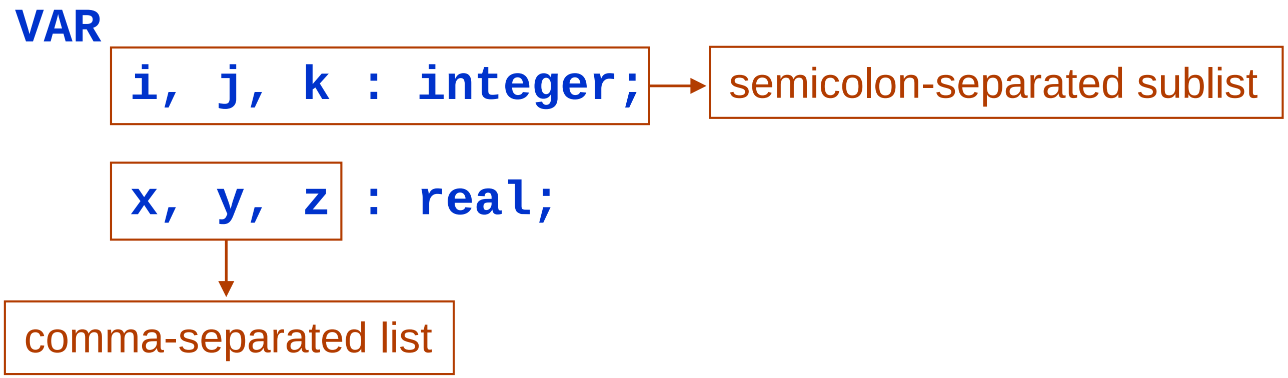


variable or field declaration



VariableDeclarationsParser.parse()

- Repeatedly call `parseIdentifierSublist()` to parse the semicolon-separated sublists of variables.
  - Loop to parse the comma-separated list of variable names.



```
VAR
    i, j, k : integer;
    x, y, z : real;
```

VariableDeclarationsParser.parse()

- Repeatedly call **parseIdentifierSublist()** to parse the semicolon-separated sublists of variables *(cont’d)*.
  - Call **parseIdentifier()** to parse each variable name.
    - Enter each identifier into the current symbol table (the one at the top of the symbol table stack).
    - Set each identifier’s definition to **VARIABLE**

```
VAR
    i, j, k : integer;
    x, y, z : real;
```

VariableDeclarationsParser.parse()

- Repeatedly call **parseIdentifierSublist()** to parse the semicolon-separated sublists of variables *(cont'd)*.
  - Call **parseTypeSpec()** to parse the type specification.
    - Consume the **:** token.
    - Call **TypeSpecificationParser.parse()** to parse the type specification.
  - Assign the type specification to each variable in the list.

```
VAR
    i, j, k : integer;
    x, y, z : real;
```

Demo

- Pascal Cross-Referencer II
  - Parse declarations
  - Generate a detailed cross-reference listing
  - Syntax check declarations