CS 153: Concepts of Compiler Design

September 21 Class Meeting

Department of Computer Science
San Jose State University

Fall 2017

Instructor: Ron Mak

www.cs.sjsu.edu/~mak

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.

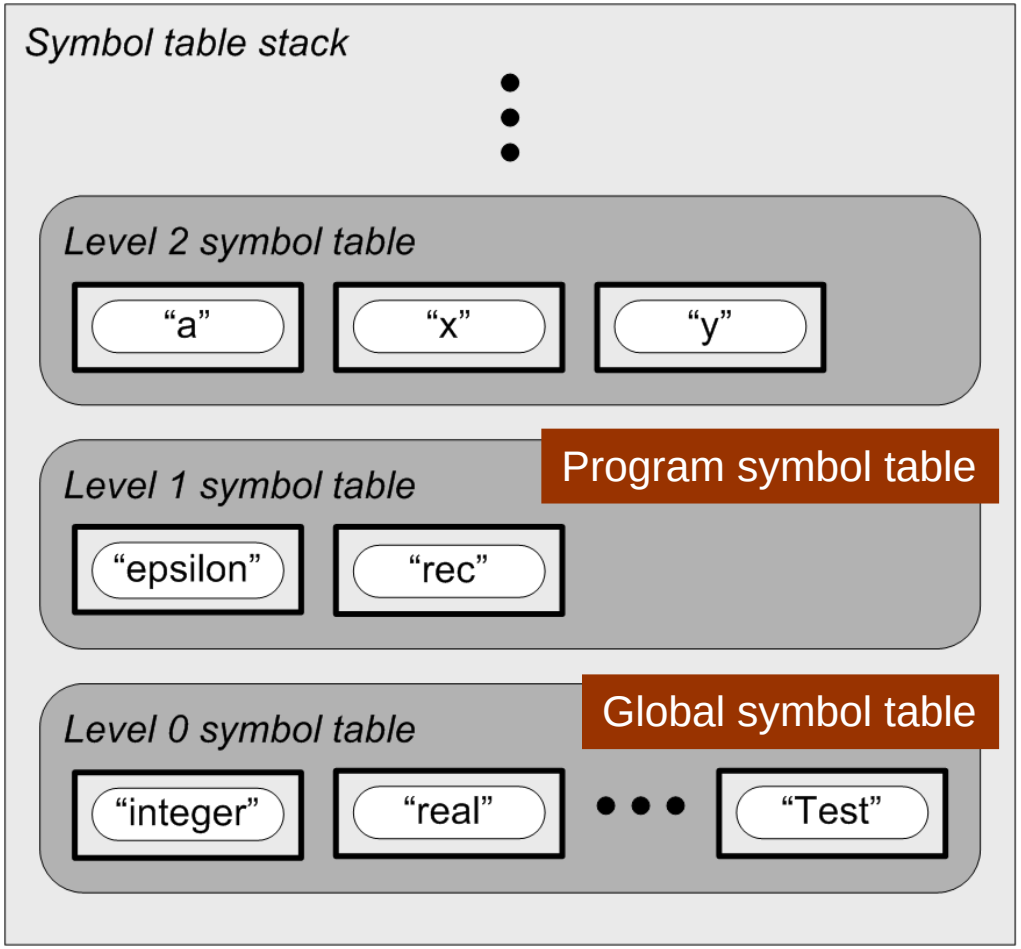
Scope and the Symbol Table Stack, *cont'd*

- 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.

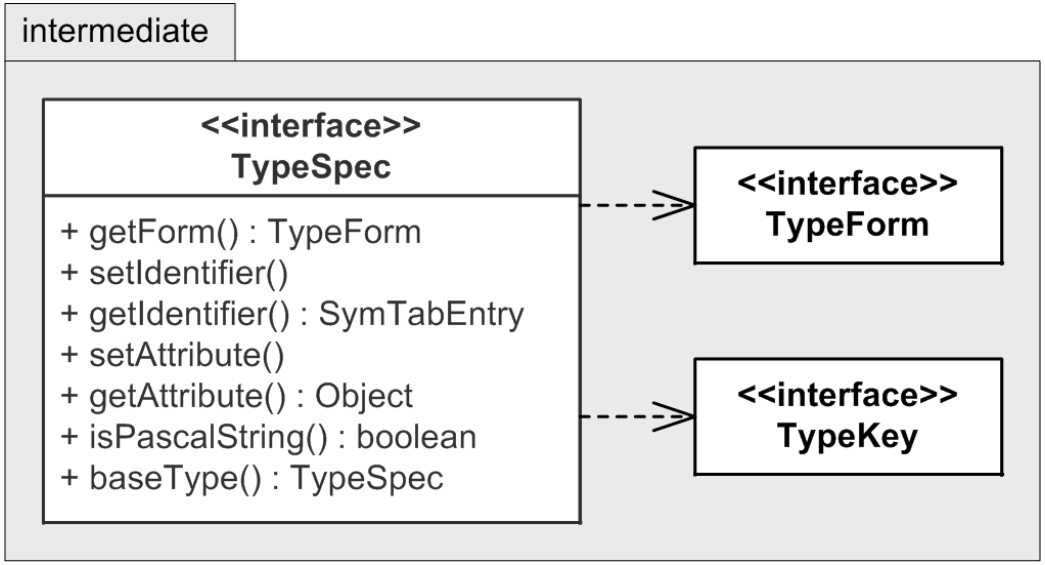
Recall that we implemented **SymTabStackImpl** as an **ArrayList<SymTab>**.

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.

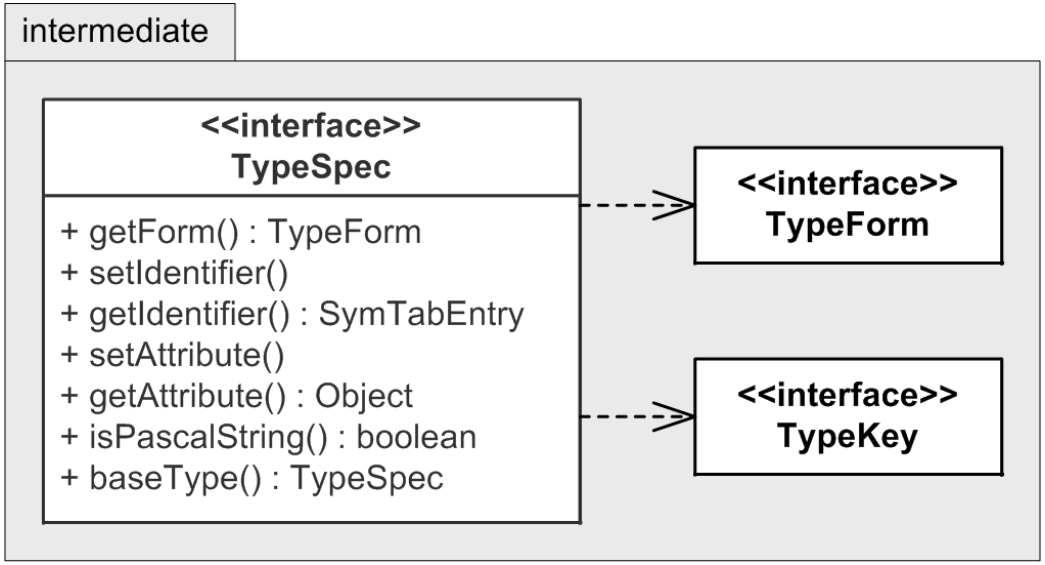
Type Specification Interfaces



Interfaces **`TypeForm`** and **`TypeKey`** are marker interfaces that define no methods.

- Conceptually, a type specification is
  - a form (scalar, array, record, etc.)
  - a type identifier (if it's named)
  - a collection of attributes

Type Specification Interfaces



- A named type refers to its type identifier: **`getIdentifier()`**
- Each type specification has certain attributes depending on its form.

- Interface **`TypeSpec`** represents a type specification.
- Method **`getForm()`** returns the type's form.
  - scalar
  - enumeration
  - subrange
  - array
  - record

Type Specification Attributes

- Each type specification has certain attributes depending on its form.

Type form	Type specification attributes
Scalar	None
Enumeration	List of enumeration constant identifiers
Subrange	Base type
	Minimum value
	Maximum value
Array	Index type
	Element type
	Element count
Record	A separate symbol table for the field identifiers

TypeFormImpl

- **TypeFormImpl** enumerated values represent the type forms.
  - The left column of the table.

```
public enum TypeFormImpl implements TypeForm
{
    SCALAR, ENUMERATION, SUBRANGE, ARRAY, RECORD;

    ...
}
```

Type form	Type specification attributes
Scalar	None
Enumeration	List of enumeration constant identifiers
Subrange	Base type
	Minimum value
	Maximum value
Array	Index type
	Element type
	Element count
Record	A separate symbol table for the field identifiers

TypeKeyImpl

- **TypeKeyImpl** enumerated values are the attribute keys.
  - The right column of the table.

Type form	Type specification attributes
Scalar	None
Enumeration	List of enumeration constant identifiers
Subrange	Base type
	Minimum value
	Maximum value
Array	Index type
	Element type
	Element count
	A separate symbol table for the field identifiers

```
public enum TypeKeyImpl
    implements TypeKey
{
    // Enumeration
    ENUMERATION_CONSTANTS,

    // Subrange
    SUBRANGE_BASE_TYPE, SUBRANGE_MIN_VALUE, SUBRANGE_MAX_VALUE,

    // Array
    ARRAY_INDEX_TYPE, ARRAY_ELEMENT_TYPE, ARRAY_ELEMENT_COUNT,

    // Record
    RECORD_SYMTAB
}
```

A **TypeFormImpl** enumerated value tells us which set of **TypeKeyImpl** attribute keys to use.

TypeSpecImpl

- Implement a type specification as a hash map.
  - Similar to a symbol table entry and a parse tree node.

```
public class TypeSpecImpl
    extends HashMap<TypeKey, Object>
    implements TypeSpec
{
    private TypeForm form;           // type form
    private SymTabEntry identifier;  // type identifier

    public TypeSpecImpl(TypeForm form)
    {
        this.form = form;
        this.identifier = null;
    }
    ...
}
```

`TypeSpecImpl` String Constructor

- A constructor to make Pascal string types.

```
public TypeSpecImpl(String value)
{
    this.form = ARRAY;

    TypeSpec indexType = new TypeSpecImpl(SUBRANGE);
    indexType.setAttribute(SUBRANGE_BASE_TYPE, Predefined.integerType);
    indexType.setAttribute(SUBRANGE_MIN_VALUE, 1);
    indexType.setAttribute(SUBRANGE_MAX_VALUE, value.length());

    setAttribute(ARRAY_INDEX_TYPE, indexType);
    setAttribute(ARRAY_ELEMENT_TYPE, Predefined.charType);
    setAttribute(ARRAY_ELEMENT_COUNT, value.length());
}
```

You'll see soon where **`Predefined.integerType`** and **`Predefined.charType`** are defined.

Type form	Type specification attributes
Scalar	None
Enumeration	List of enumeration constant identifiers
Subrange	Base type
	Minimum value
	Maximum value
Array	Index type
	Element type
	Element count
Record	A separate symbol table for the field identifiers

Method `TypeSpecImpl.baseType()`

- Return the base type of a subrange type.
  - Example: Return integer for a subrange of integer.
  - Just return the type if it's not a subrange.

```
public TypeSpec baseType()
{
    return form == SUBRANGE ? (TypeSpec) getAttribute(SUBRANGE_BASE_TYPE)
                            : this;
}
```

Method `TypeSpecImpl.isPascalString()`

- Is this a string type?

```
public boolean isPascalString()
{
    if (form == ARRAY) {
        TypeSpec elmtType  = (TypeSpec) getAttribute(ARRAY_ELEMENT_TYPE);
        TypeSpec indexType = (TypeSpec) getAttribute(ARRAY_INDEX_TYPE);

        return (elmtType.baseType()  == Predefined.charType) &&
               (indexType.baseType() == Predefined.integerType);
    }
    else {
        return false;
    }
}
```

To be a Pascal string type,
the **index type** must be **integer**
and the **element type** must be **char**.

Type Factory

- A type factory creates type specifications.
  - In package **`intermediate`**.
- Two factory methods.
  - Pascal string types.
  - All other types.

```
public class TypeFactory
{
    public static TypeSpec createType(TypeForm form)
    {
        return new TypeSpecImpl(form);
    }

    public static TypeSpec createStringType(String value)
    {
        return new TypeSpecImpl(value);
    }
}
```

How are Identifiers Defined?

- Identifiers are no longer only the names of variables.
- Class **`DefinitionImpl`** enumerates all the ways an identifier can be defined.
  - Package **`intermediate.symtabimpl`**.

```
public enum DefinitionImpl implements Definition
{
    CONSTANT, ENUMERATION_CONSTANT("enumeration constant"),
    TYPE, VARIABLE, FIELD("record field"),
    VALUE_PARM("value parameter"), VAR_PARM("VAR parameter"),
    PROGRAM_PARM("program parameter"),
    PROGRAM, PROCEDURE, FUNCTION,
    UNDEFINED;
    ...
}
```

Think of **`Definition`** as the role an identifier plays.

Predefined Scalar Types

- Initialized by class **`Predefined`** in the global scope.
  - Package **`intermediate.symtabimpl`**.
  - Example: **`integer`**

```
public static TypeSpec integerType;
public static SymTabEntry integerId;
...
private static void initializeTypes(SymTabStack symTabStack)
{
    integerId = symTabStack.enterLocal("integer");
    integerType = TypeFactory.createType(SCALAR);
    integerType.setIdentifier(integerId);
    integerId.setDefinition(DefinitionImpl.TYPE);
    integerId.setTypeSpec(integerType);
    ...
}
```

Type specification **`integerType`** and the symbol table entry **`integerId`** refer to each other.

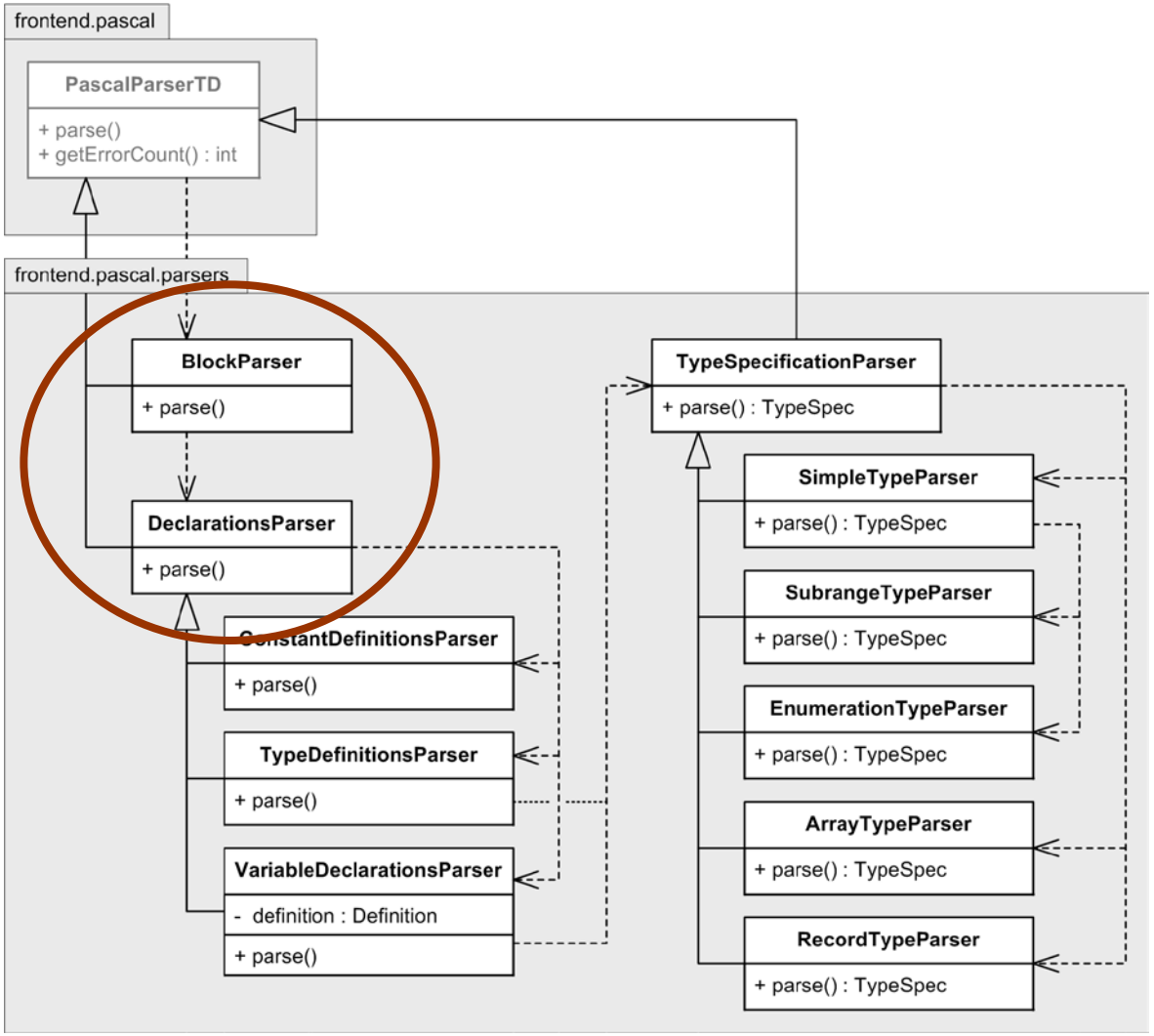
Predefined Scalar Types

- Predefined type **`boolean`** introduces two predefined constants, **`true`** and **`false`**.

```
falseId = symTabStack.enterLocal("false");
falseId.setDefinition(DefinitionImpl.ENUMERATION_CONSTANT);
falseId.setTypeSpec(booleanType);
falseId.setAttribute(CONSTANT_VALUE, new Integer(0));

trueId = symTabStack.enterLocal("true");
trueId.setDefinition(DefinitionImpl.ENUMERATION_CONSTANT);
trueId.setTypeSpec(booleanType);
trueId.setAttribute(CONSTANT_VALUE, new Integer(1));

ArrayList<SymTabEntry> constants = new ArrayList<SymTabEntry>();
constants.add(falseId);
constants.add(trueId);
booleanType.setAttribute(ENUMERATION_CONSTANTS, constants);
```

Add the constant identifiers to the **`ENUMERTION_CONSTANTS`** list attribute.

Parsing Pascal Declarations



Method **PascalParserTD.parse()**

```
symTabStack = SymTabFactory.createSymTabStack();
iCode = ICodeFactory.createICode();
errorHandler = new PascalErrorHandler();

Predefined.initialize(symTabStack);

routineId = symTabStack.enterLocal("DummyProgramName".toLowerCase());
routineId.setDefinition(DefinitionImpl.PROGRAM);
symTabStack.setProgramId(routineId);

routineId.setAttribute(ROUTINE_SYMTAB, symTabStack.push());
routineId.setAttribute(ROUTINE_ICODE, iCode);

Token token = nextToken();

BlockParser blockParser = new BlockParser(this);
blockParser.parse(token);
symTabStack.pop();
```

Create a new symbol table stack (includes the **level 0** global symbol table).

Predefined types ➔ global symbol table.

Push the new **level 1** symbol table.

The symbol table entry of the program id keeps a pointer to the level 1 symbol table.

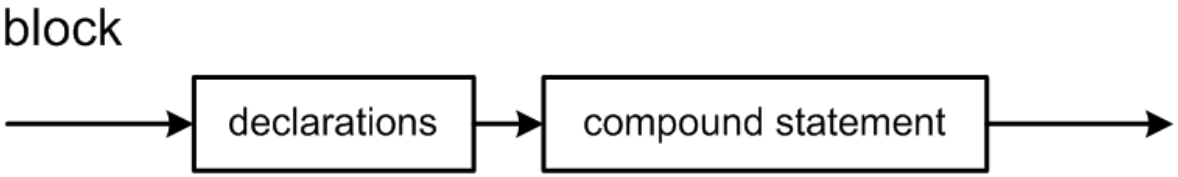
Parse the program's **block**.
When done, pop off the level 1 symbol table.

Method **PascalParserTD.parse()**

```
token = currentToken();
if (token.getType() != DOT) {
    errorHandler.flag(token, MISSING_PERIOD, this);
}
token = currentToken();

float elapsedTime = (System.currentTimeMillis() - startTime)/1000f;
sendMessage(new Message(PARSER_SUMMARY,
                        new Number[] {token.getLineNumber(),
                                      getErrorCount(),
                                      elapsedTime}));
```

Parse the final period.

Method `BlockParser.parse()`



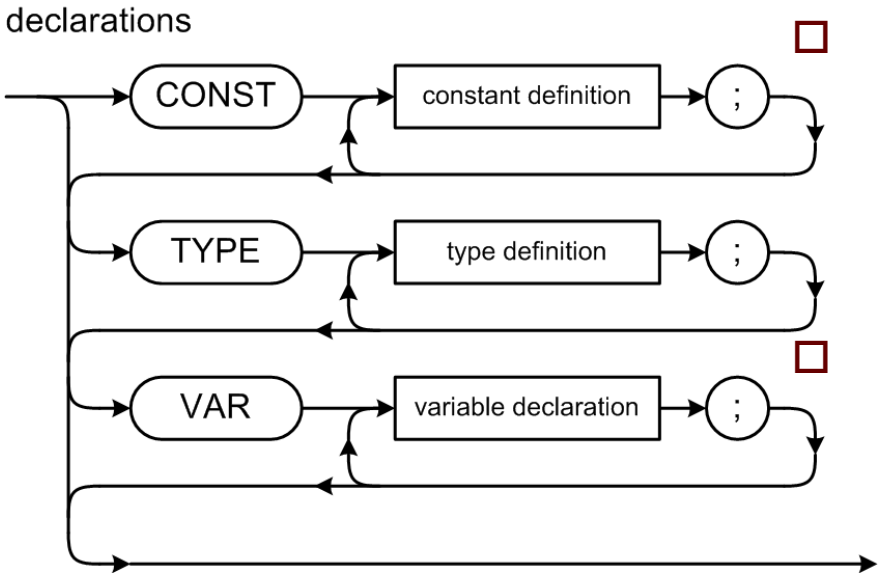

```
DeclarationsParser declarationsParser = new DeclarationsParser(this);
StatementParser statementParser = new StatementParser(this);

// Parse any declarations.
declarationsParser.parse(token);

token = synchronize(StatementParser.STMT_START_SET);
TokenType tokenType = token.getType();
ICodeNode rootNode = null;

// Look for the BEGIN token to parse a compound statement.
if (tokenType == BEGIN) {
    rootNode = statementParser.parse(token);
}
...
return rootNode;
```

Parse declarations.

Synchronize to the start of a statement.

Parse the compound statement.

Method `DeclarationsParser.parse()`



- Parse constant definitions.
  - If there is the **`CONST`** reserved word.
  - Call the **`parse()`** method of **`ConstantDefinitionsParser`**
- Parse type definitions.
  - If there is the **`TYPE`** reserved word.
  - Call the **`parse()`** method of **`TypeDefinitionsParser`**
- Parse variable declarations.
  - If there is the **`VAR`** reserved word.
  - Call the **`parse()`** method of **`VariableDeclarationsParser`**

ConstantDefinitionsParser

```
CONST
    epsilon = 1.0e-6;
    pi = 3.1415926;
    delta = epsilon;
```

parse()

parseConstant()

parseIdentifierConstant()

getConstantType()

Method **ConstantDefinitionsParser.parse()**

```
token = synchronize(IDENTIFIER_SET);

while (token.getType() == IDENTIFIER) {
    String name = token.getText().toLowerCase();
    SymTabEntry constantId = symTabStack.lookupLocal(name);

    if (constantId == null) {
        constantId = symTabStack.enterLocal(name);
        constantId.appendLineNumber(token.getLineNumber());
    }
    else {
        errorHandler.flag(token, IDENTIFIER_REDEFINED, this);
        constantId = null;
    }

    token = nextToken();  // consume the identifier token

    token = synchronize(EQUALS_SET);
    if (token.getType() == EQUALS) {
        token = nextToken();  // consume the =
    }
```

Loop over the constant definitions.

Enter the **constant identifier** into the symbol table but don't set its definition to **CONSTANT** yet.

Synchronize and parse the **=** sign.

continued …

Method **ConstantDefinitionsParser.parse()**

```
    Token constantToken = token;
    Object value = parseConstant(token);

    if (constantId != null) {
        constantId.setDefinition(CONSTANT);
        constantId.setAttribute(CONSTANT_VALUE, value);

        TypeSpec constantType =
            constantToken.getType() == IDENTIFIER
                ? getConstantType(constantToken)
                : getConstantType(value);
        constantId.setTypeSpec(constantType);
    }

    ...

    token = synchronize(IDENTIFIER_SET);
} // while
```

Parse the constant's **value**.

Now set the **constant identifier** to **CONSTANT**.

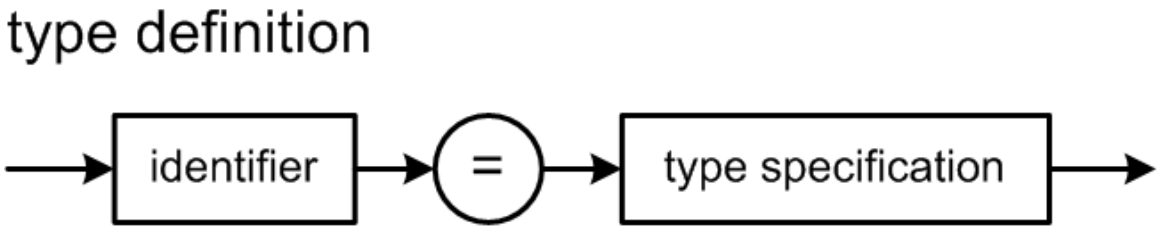
Set the constant's value.

Set the constant's **type**.

Guard against the constant definition **pi = pi**

Synchronize at the start of the next constant definition.

Review: Type Definitions

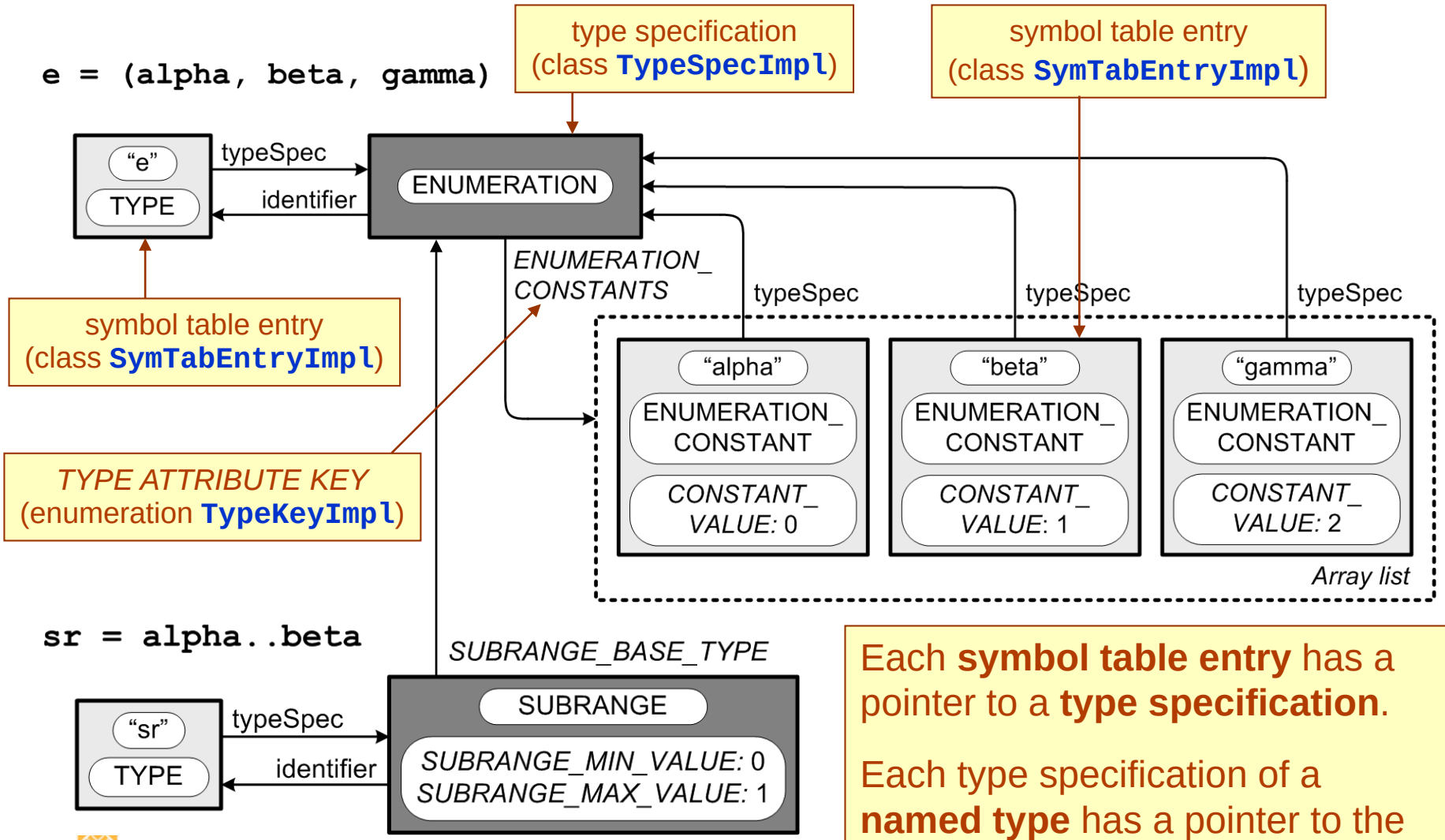


- Sample type definitions:

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

How can we represent **type specification information** in our symbol table and associate the correct type specification with an identifier?

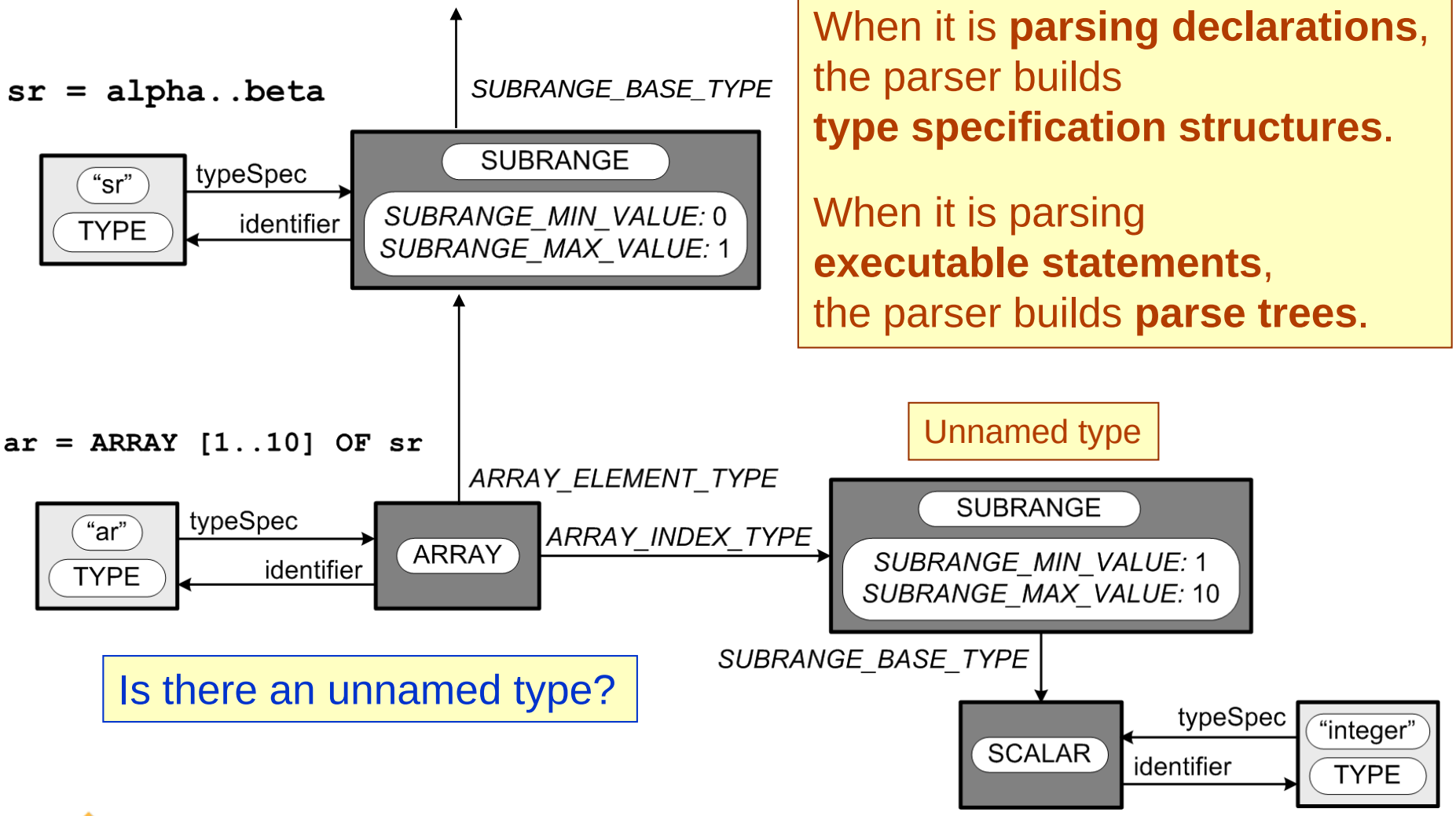
Type Definition Structures



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*

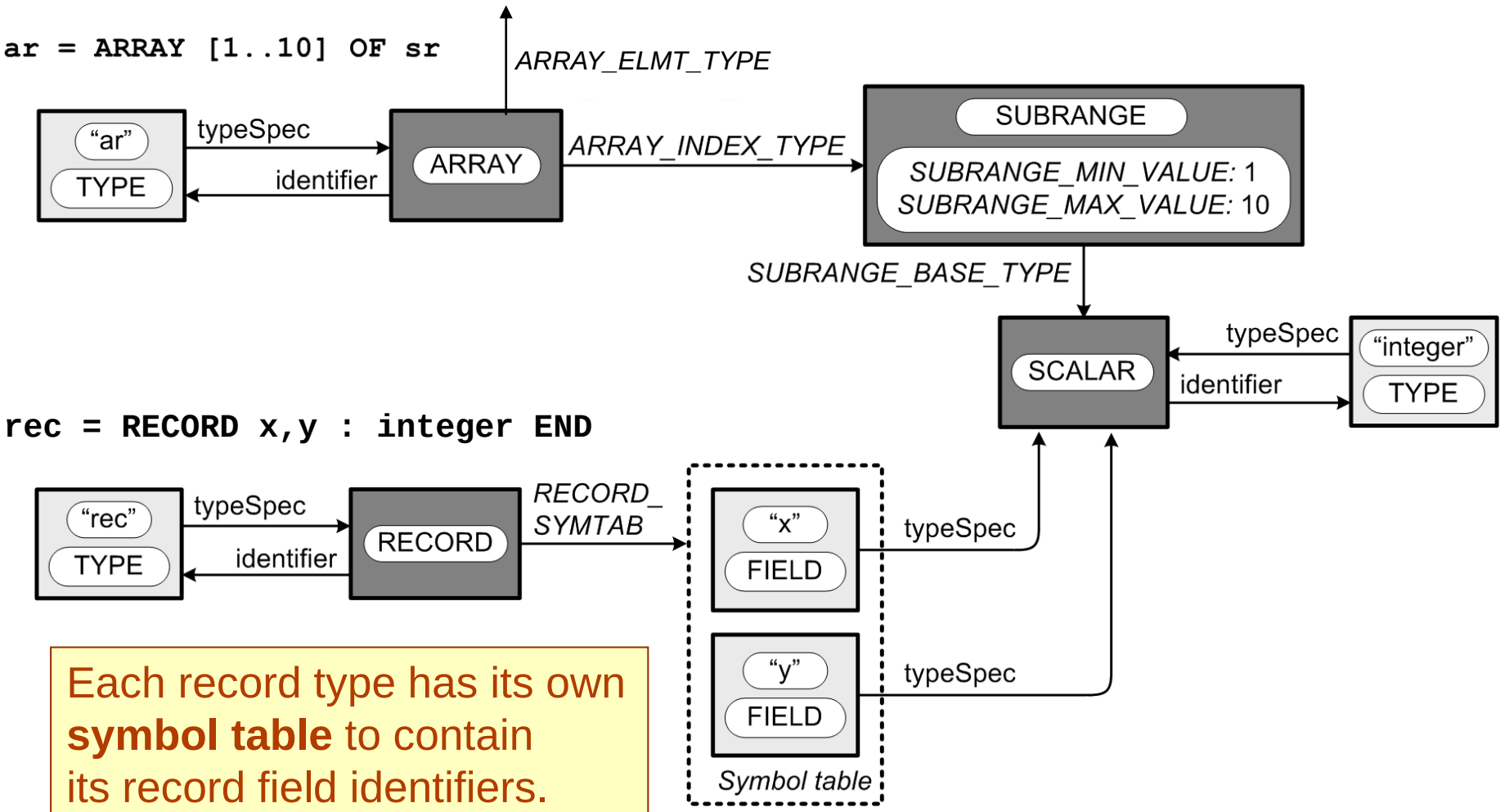


When it is **parsing declarations**, the parser builds **type specification structures**.

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

Is there an unnamed type?

Type Definition Structures, *cont'd*



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