CMPE 152: Compiler Design

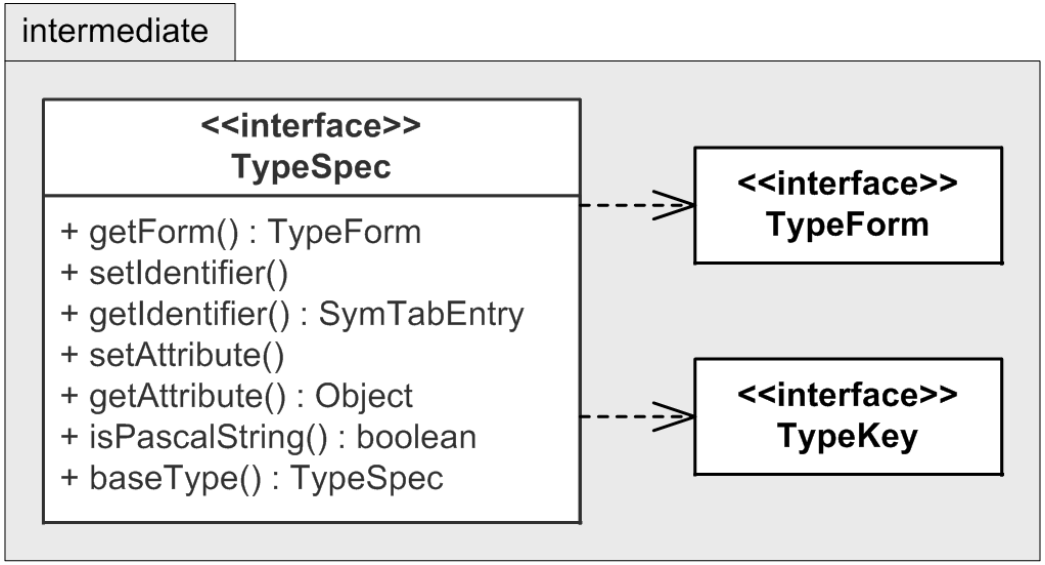
September 21 Class Meeting

Department of Computer Engineering
San Jose State University

Fall 2017
Instructor: Ron Mak

www.cs.sjsu.edu/~mak

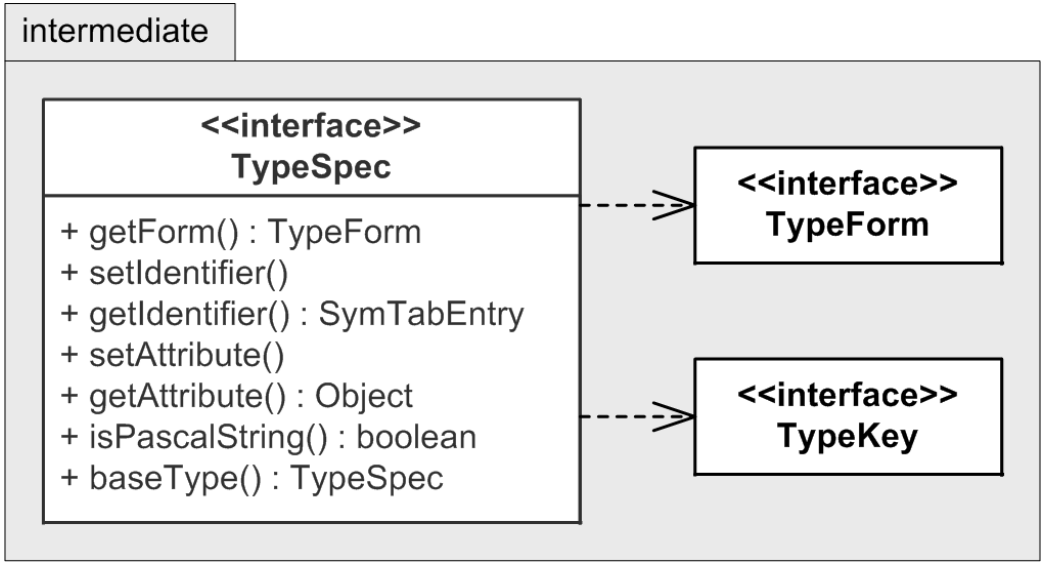
Type Specification Interfaces



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: **`get_identifier()`**
- 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.

```
enum class TypeFormImpl
{
    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

```
enum class TypeKeyImpl
{
    // 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 with a map.
  - Similar to a symbol table entry and a parse tree node.

```
class TypeSpecImpl : public TypeSpec
{
public:
    TypeSpecImpl();
    TypeSpecImpl(TypeForm form);
    ...

private:
    TypeForm form;
    SymTabEntry *type_id;  // type identifier
    map<TypeKey, TypeValue *> contents;
    ...
}
```

`TypeSpecImpl` String Constructor

A constructor to make a Pascal string.

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

```
TypeSpecImpl::TypeSpecImpl(string value)
{
    initialize();
    form = (TypeForm) TF_ARRAY;

    TypeSpec *index_type = new TypeSpecImpl((TypeForm) TF_SUBRANGE);
    index_type->set_attribute((TypeKey) SUBRANGE_BASE_TYPE,
                              new TypeValue(Predefined::integer_type));
    index_type->set_attribute((TypeKey) SUBRANGE_MIN_VALUE,
                              new TypeValue(new DataValue(1)));
    int length = value.length();
    index_type->set_attribute((TypeKey) SUBRANGE_MAX_VALUE,
                              new TypeValue(new DataValue(length)));

    this->set_attribute((TypeKey) ARRAY_INDEX_TYPE,
                        new TypeValue(index_type));
    this->set_attribute((TypeKey) ARRAY_ELEMENT_TYPE,
                        new TypeValue(Predefined::char_type));
    this->set_attribute((TypeKey) ARRAY_ELEMENT_COUNT,
                        new TypeValue(new DataValue(length)));
}
```

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.

```
TypeSpec *TypeSpecImpl::base_type()
{
    if (form == (TypeForm) TF_SUBRANGE)
    {
        TypeValue *type_value =
            this->get_attribute((TypeKey) SUBRANGE_BASE_TYPE);
        return type_value->typespec;
    }
    else return this;
}
```

TypeSpecImpl::is_pascal_string()

- Is this a string type?

```
bool TypeSpecImpl::is_pascal_string()
{
    if (form == (TypeForm) TF_ARRAY)
    {
        TypeValue *type_value = this->get_attribute((TypeKey) ARRAY_ELEMENT_TYPE);
        TypeSpec  *elmt_type  = type_value->typespec;


        type_value = this->get_attribute((TypeKey) ARRAY_INDEX_TYPE);
        TypeSpec *index_type = type_value->typespec;


        return (elmt_type->base_type()  == Predefined::char_type) &&
               (index_type->base_type() == Predefined::integer_type);
    }
    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 namespace **`intermediate`**.
- Two factory methods.
  - Pascal string types.
  - All other types.

```
TypeSpec *TypeFactory::create_type(const TypeForm form)
{
    return new TypeSpecImpl(form);
}

TypeSpec *TypeFactory::create_string_type(const 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.
  - Namespace **`intermediate::symtabimpl`**.

```
enum class DefinitionImpl
{
    CONSTANT, ENUMERATION_CONSTANT,
    TYPE, VARIABLE, FIELD,
    VALUE_PARM, VAR_PARM,
    PROGRAM_PARM,
    PROGRAM, PROCEDURE, FUNCTION,
    UNDEFINED,
};
```

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

Predefined Scalar Types

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

```
TypeSpec *Predefined::integer_type;
SymTabEntry *Predefined::integer_id;
...
void Predefined::initialize_types(SymTabStack *symtab_stack)
{
    integer_id = symtab_stack->enter_local("integer");
    integer_type = TypeFactory::create_type((TypeForm) TF_SCALAR);
    integer_type->set_identifier(integer_id);
    integer_id->set_definition((Definition) DF_TYPE);
    integer_id->set_typespec(integer_type);
    ...
}
```

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

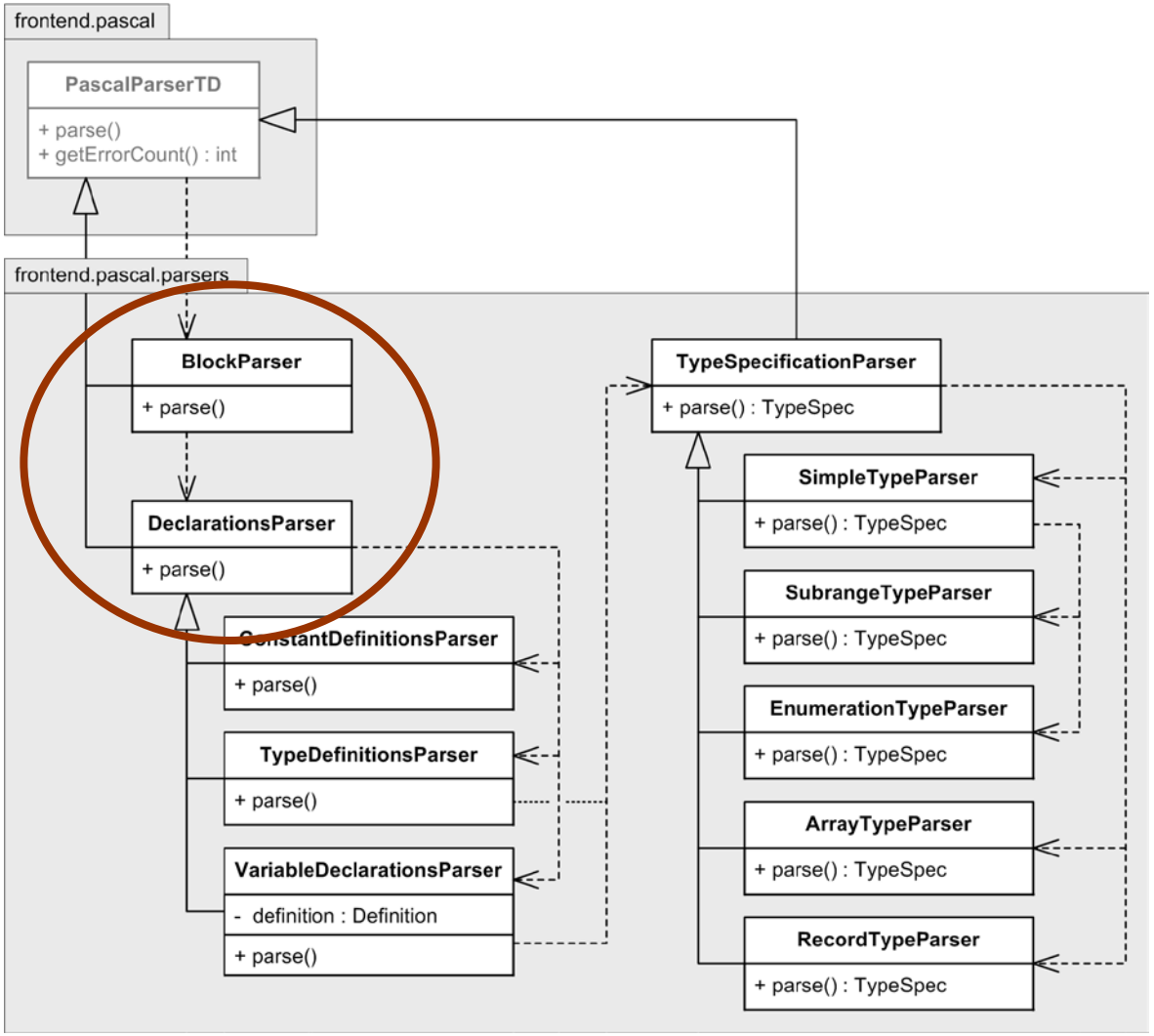
Predefined Scalar Types

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

```
false_id = symtab_stack->enter_local("false");
false_id->set_definition((Definition) DF_ENUMERATION_CONSTANT);
false_id->set_typespec(boolean_type);
false_id->set_attribute((SymTabKey) CONSTANT_VALUE,
                        new EntryValue(new DataValue(0)));

true_id = symtab_stack->enter_local("true");
true_id->set_definition((Definition) DF_ENUMERATION_CONSTANT);
true_id->set_typespec(boolean_type);
true_id->set_attribute((SymTabKey) CONSTANT_VALUE,
                       new EntryValue(new DataValue(1)));

TypeValue *type_value = new TypeValue();
type_value->v.push_back(false_id);
type_value->v.push_back(true_id);
boolean_type->set_attribute((TypeKey) ENUMERATION_CONSTANTS,
                            type_value);
```

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

Parsing Pascal Declarations



PascalParserTD::parse()

```
icode = ICodeFactory::create_icode();
Predefined::initialize(symtab_stack);

routine_id = symtab_stack->enter_local("dummyprogramname");
routine_id->set_definition((Definition) DefinitionImpl::PROGRAM);
symtab_stack->set_program_id(routine_id);

routine_id->set_attribute((SymTabKey) ROUTINE_SYMTAB,
                          new EntryValue(symtab_stack->push()));
routine_id->set_attribute((SymTabKey) ROUTINE_ICODE,
                          new EntryValue(icode));

Token *token = next_token(nullptr);

BlockParser block_parser(this);
ICodeNode *root_node = block_parser.parse_block(token, routine_id);
icode->set_root(root_node);
symtab_stack->pop();
```

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.

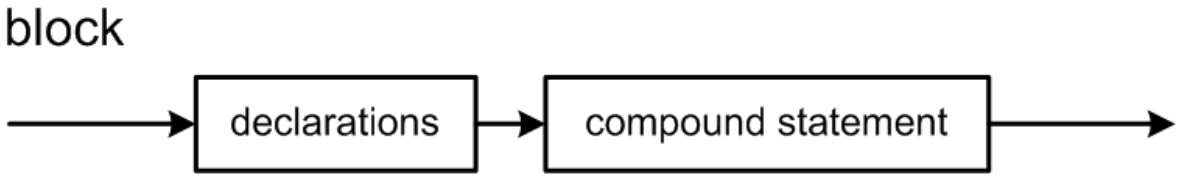
Parse the program's **block**.
When done, pop off the level 1 symbol table.

PascalParserTD::parse(), *cont'd*

```
token = current_token();
if (token->get_type() != (TokenType) PT_DOT)
{
    error_handler.flag(token, MISSING_PERIOD, this);
}

int last_line_number = token->get_line_number();

if (root_node != nullptr) icode->set_root(root_node);

steady_clock::time_point end_time = steady_clock::now();
double elapsed_time =
        duration_cast<duration<double>>(end_time - start_time).count();
Message message(PARSER_SUMMARY,
                LINE_COUNT, to_string(last_line_number),
                ERROR_COUNT, to_string(get_error_count()),
                ELAPSED_TIME, to_string(elapsed_time));
send_message(message);
```

Parse the final period.

BlockParser::parse()



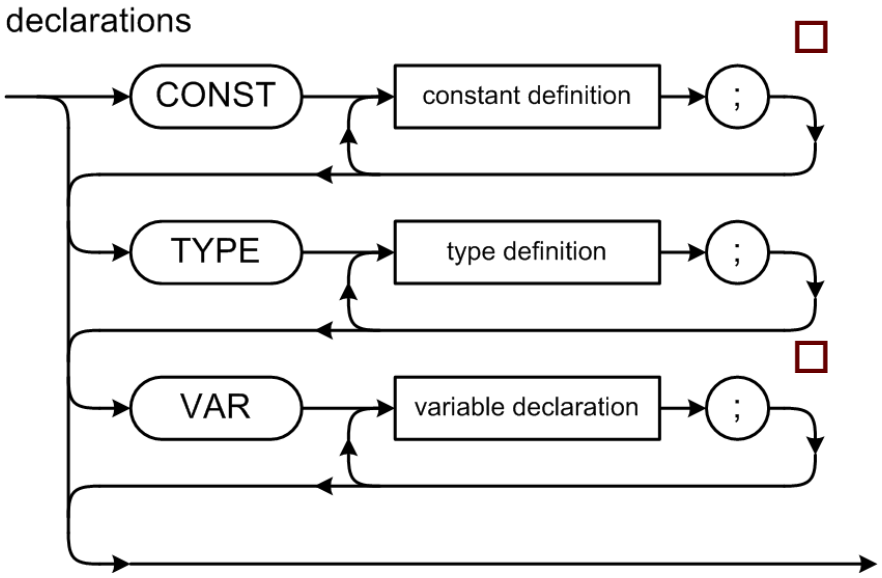

```
ICodeNode *BlockParser::parse_block(Token *token, SymTabEntry *routine_id)
    throw (string)
{
    DeclarationsParser declarations_parser(this);
    StatementParser statement_parser(this);

    declarations_parser.parse_declaration(token);

    token = synchronize(StatementParser::STMT_START_SET);
    TokenType token_type = token->get_type();
    ICodeNode *root_node = nullptr;

    if (token_type == (TokenType) PT_BEGIN)
    {
        root_node = statement_parser.parse_statement(token);
    }
...
return root_node;
```

Parse declarations.

Synchronize to the start of a statement.

Parse the compound statement.

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()

ConstantDefinitionsParser::parse()

```
void ConstantDefinitionsParser::parse_declaration(Token *token) throw (string)
{
    token = synchronize(IDENTIFIER_SET);

    while (token->get_type() == (TokenType) PT_IDENTIFIER)
    {
        string name = token->get_text();
        transform(name.begin(), name.end(), name.begin(), ::tolower);
        SymTabEntry *constant_id = symtab_stack->lookup_local(name);

        if (constant_id == nullptr)
        {
            constant_id = symtab_stack->enter_local(name);
            constant_id->append_line_number(token->get_line_number());
        }
        else
        {
            error_handler.flag(token, IDENTIFIER_REDEFINED, this);
            constant_id = nullptr;
        }

        token = next_token(token);  // consume the identifier token
        token = synchronize(EQUALS_SET);
        if (token->get_type() == (TokenType) PT_EQUALS)
        {
            token = next_token(token);  // 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 *constant_token = token;
        DataValue *data_value = parse_constant(token);
        if (data_value == nullptr) data_value = new DataValue(-1);

        if (constant_id != nullptr)
        {
            constant_id->set_definition((Definition) DF_CONSTANT);
            constant_id->set_attribute((SymTabKey) CONSTANT_VALUE,
                                       new EntryValue(data_value));

            TypeSpec *constant_type =
                constant_token->get_type()
                            == (TokenType) PT_IDENTIFIER
                    ? get_constant_type(constant_token)
                    : get_constant_type(data_value);
            constant_id->set_typespec(constant_type);
        }

    ...

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

Parse the constant's **value**.

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

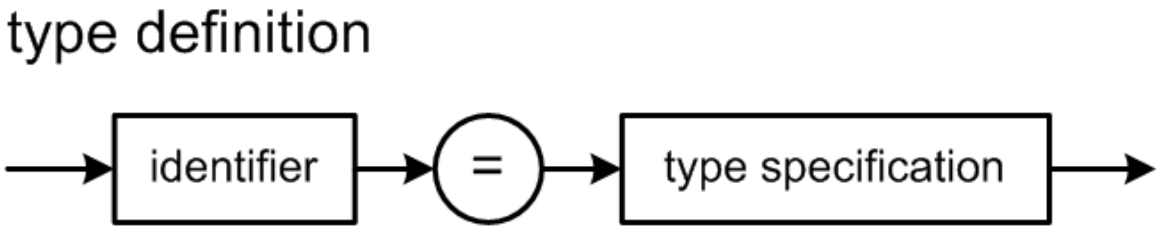
Set the constant's value.

Set the constant's **type**.

Synchronize at the start of the next constant definition.

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

Review: Type Definitions

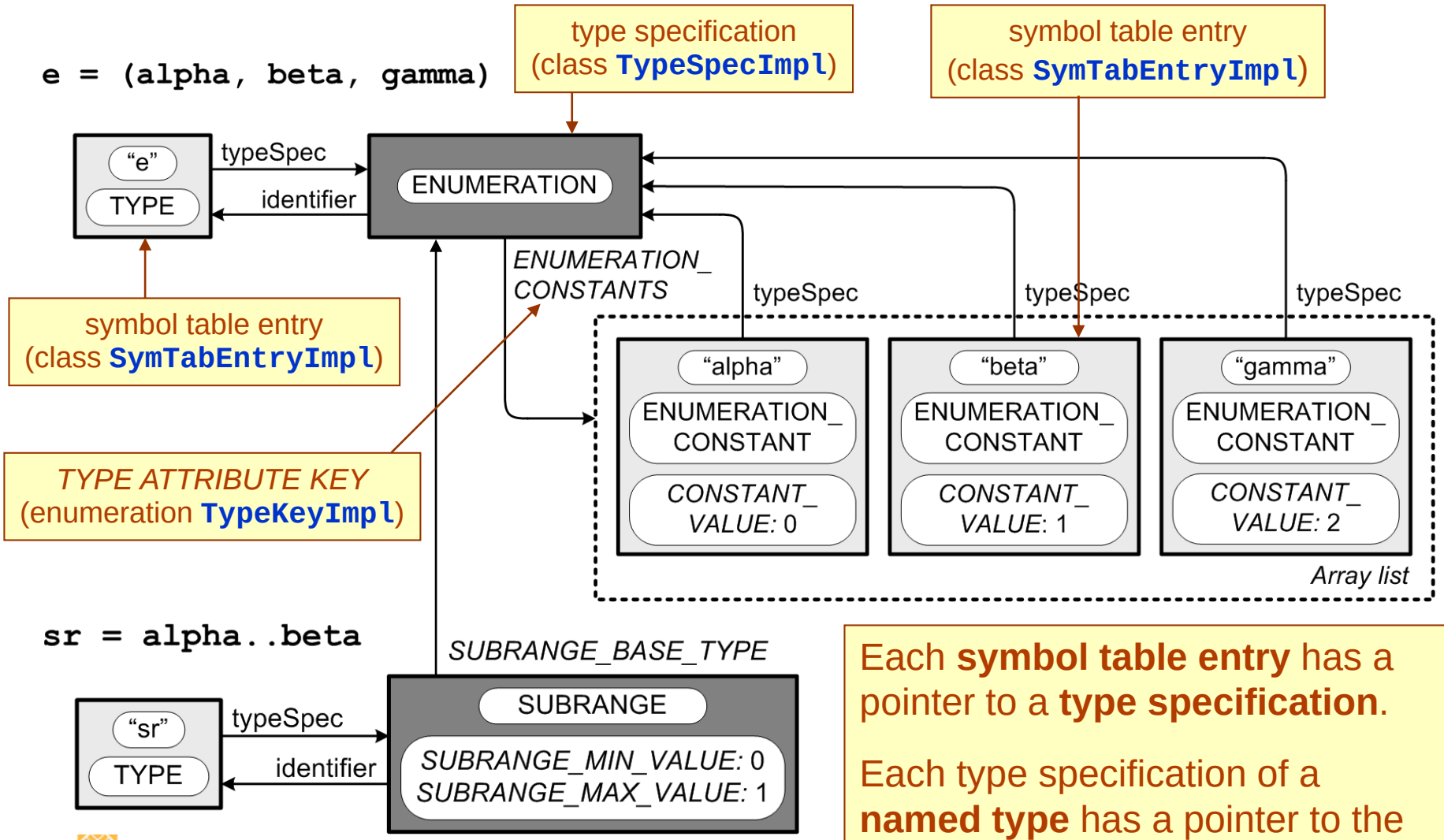


- Sample type definitions:

```
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?

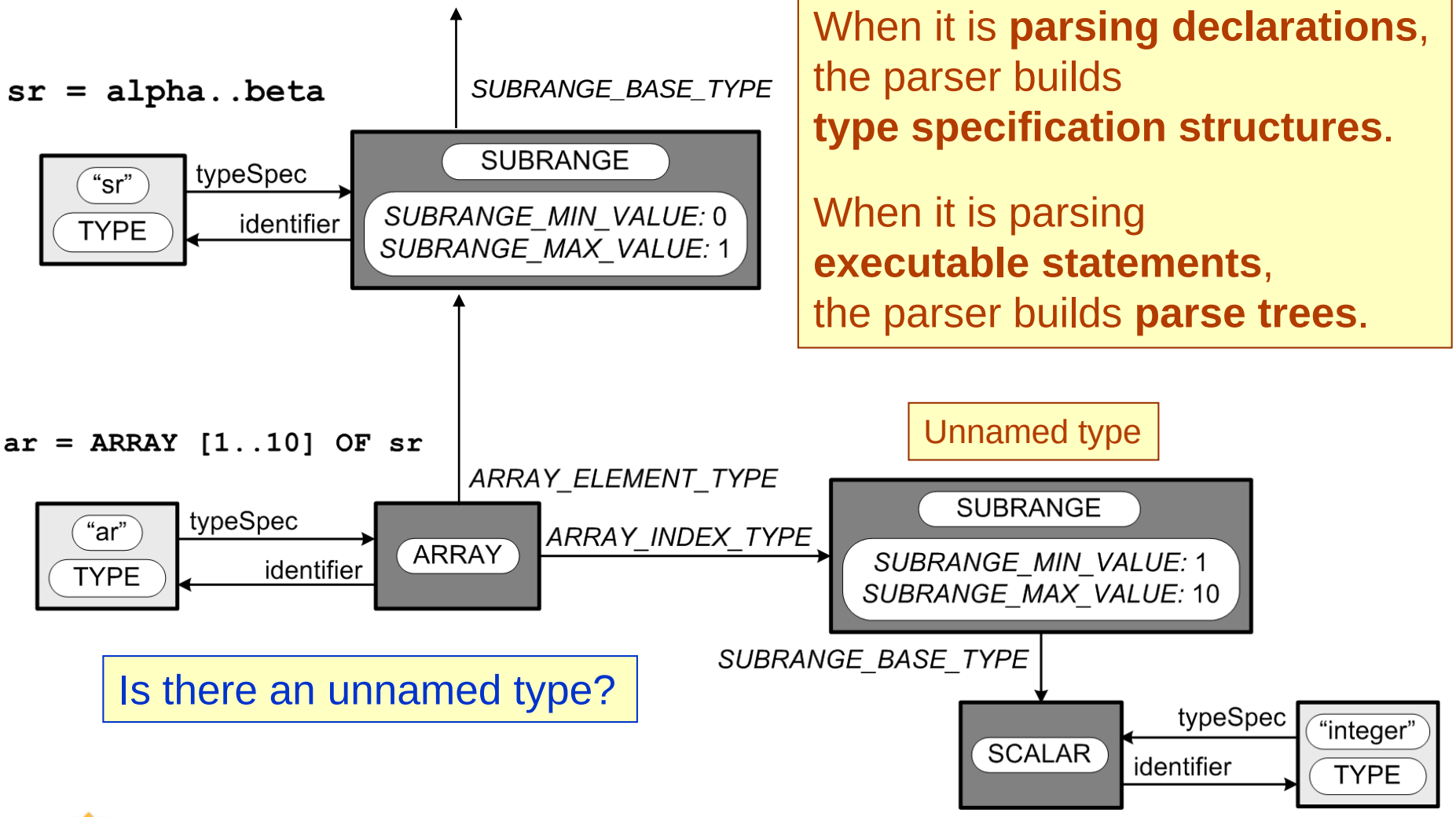
Type Definition Structures



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*

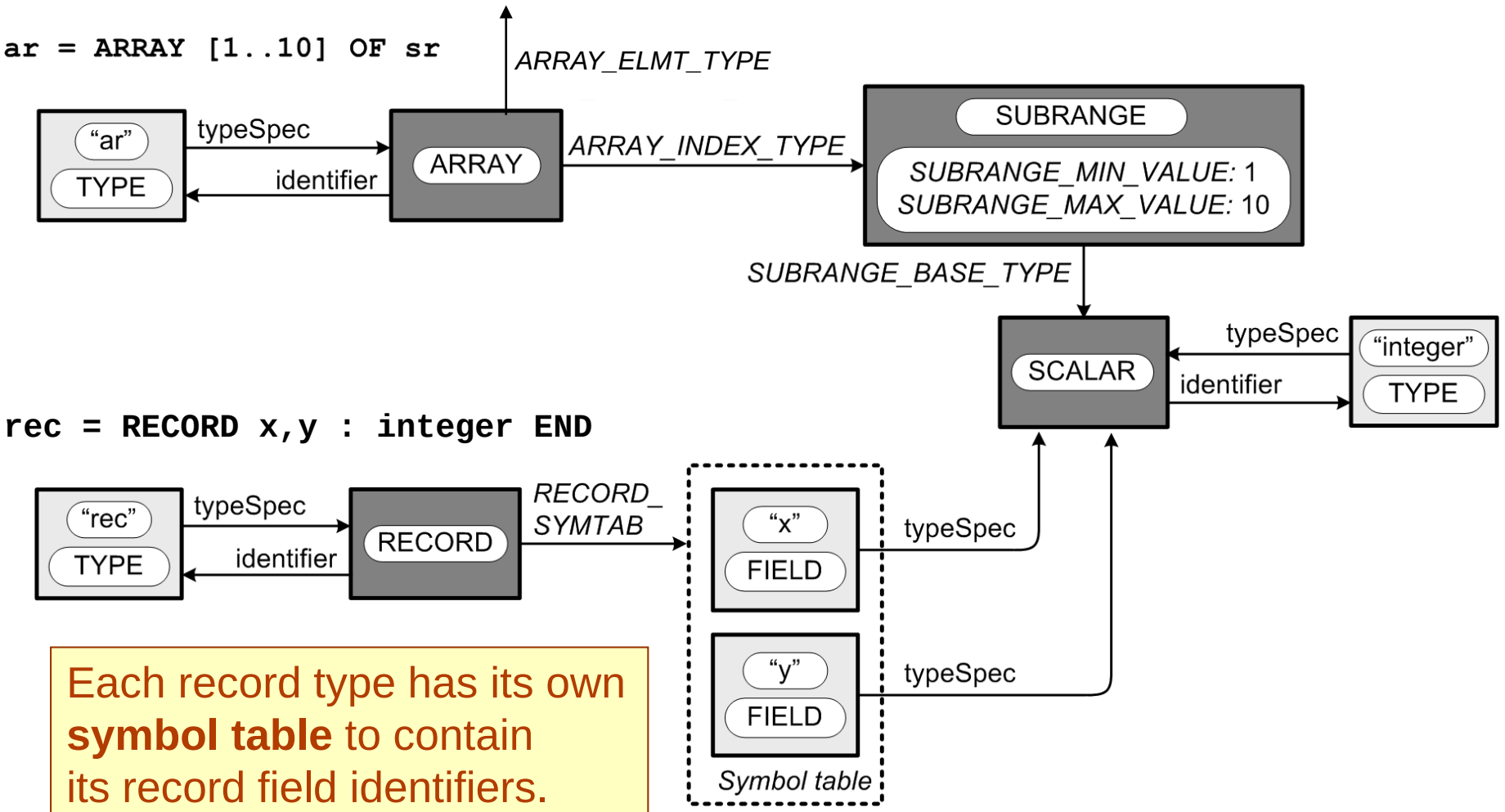


When it is **parsing declarations**, the parser builds **type specification structures**.

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.