CMPE 152: Compiler Design

September 26 Lab

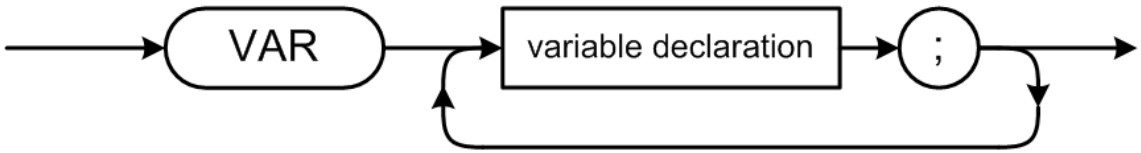
Department of Computer Engineering
San Jose State University

Fall 2017
Instructor: Ron Mak

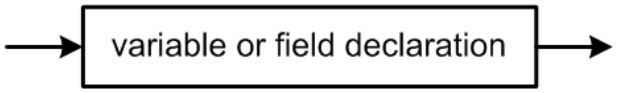
www.cs.sjsu.edu/~mak

Pascal Variable Declarations

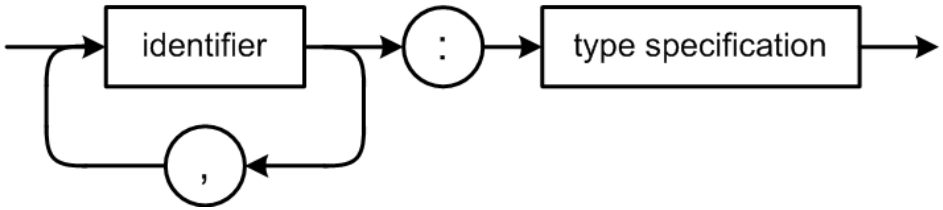
variable declarations



variable declaration

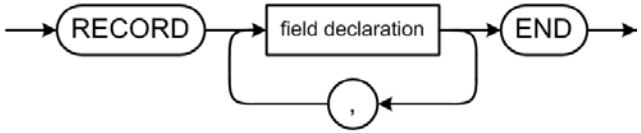


variable or field declaration

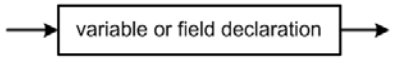


Compare to:

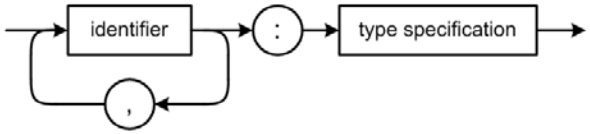
record type



field declaration

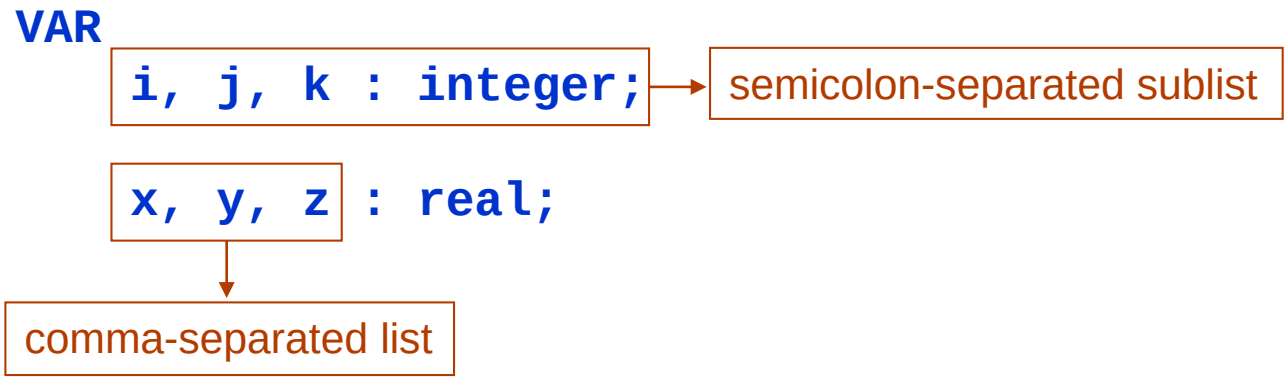


variable or field declaration



VariableDeclarationsParser.parse()

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



VariableDeclarationsParser.parse()

- Repeatedly call **parse_identifier_sublist()** to parse the semicolon-separated sublists of variables *(cont'd)*.
  - Call **parse_identifier()** 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 **parse_identifier_sublist()** 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

Now that we can parse declarations …

- We can parse variables that have subscripts and fields.

Chapter 10

  - Example:

```
var9.rec.flda[b][0,'m'].flda[d] := 'p'
```

- We can perform type checking.
  - A semantic action.

Type Checking

- Ensure that the types of the operands are type-compatible with their operator.
  - Example: You can only perform an integer division with the **`DIV`** operator and integer operands.
  - Example: The relational operators **`AND`** and **`OR`** can only be used with boolean operands.
- Ensure that a value being assigned to a variable is assignment-compatible with the variable.
  - Example: You cannot assign a string value to an integer variable.

Type Specifications and the Parse Tree

- Every Pascal expression has a data type.
- Add a type specification to every parse tree node.
  - "Decorate" the parse tree with type information.
  - In interface **ICodeNode.h**:

```
void set_typespec(TypeSpec *spec) = 0;
TypeSpec *get_typespec() const = 0;
```

  - In class **ICodeNodeImpl.h**:

```
private:
    TypeSpec *typespec;  // data type specification

public:
    void set_typespec(TypeSpec typeSpec) { ... }
    TypeSpec *get_typespec() { ... }
```

Class **`TypeChecker`**

- Static boolean methods for type checking:
  - **`is_integer()`**
  - **`are_both_integer()`**
  - **`is_real()`**
  - **`is_integer_or_real()`**
  - **`is_at_least_one_real()`**
  - **`is_boolean()`**
  - **`are_both_boolean()`**
  - **`is_char()`**
  - **`are_assignment_compatible()`**
  - **`are_comparison_cCompatible()`**
  - **`equal_length_strings()`**

In namespace **`intermediate::typeimpl`**.

Class **TypeChecker**, *cont'd*

```
bool TypeChecker::is_integer(TypeSpec *typespec)
{
    return    (typespec != nullptr)
           && (typespec->base_type() == Predefined::integer_type);
}

bool TypeChecker::are_both_integer(TypeSpec *typespec1,
                                   TypeSpec *typespec2)
{
    return is_integer(typespec1) && is_integer(typespec2);
}

...

bool TypeChecker::is_at_least_one_real(TypeSpec *typespec1,
                                       TypeSpec *typespec2)
{
    return (is_real(typespec1) && is_real(typespec2)) ||
           (is_real(typespec1) && is_integer(typespec2)) ||
           (is_integer(typespec1) && is_real(typespec2));
}
```

Assignment and Comparison Compatible

- In classic Pascal, a value is assignment-compatible with a target variable if:
  - both have the same type
  - the target is real and the value is integer
  - they are equal-length strings
- Two values are comparison-compatible (they can be compared with relational operators) if:
  - both have the same type
  - one is integer and the other is real
  - they are equal-length strings

Assignment Compatible

```
bool TypeChecker::are_assignment_compatible(TypeSpec *target_typespec,
                                            TypeSpec *value_typespec)
{
    if ((target_typespec == nullptr) || (value_typespec == nullptr))
    {
        return false;
    }

    target_typespec = target_typespec->base_type();
    value_typespec  = value_typespec->base_type();

    bool compatible = false;

    if (target_typespec == value_typespec)
    {
        compatible = true;
    }

    else if (is_real(target_typespec) && is_integer(value_typespec))
    {
        compatible = true;
    }

    else {
        compatible =    target_typespec->is_pascal_string()
                     && value_typespec->is_pascal_string();
    }

    return compatible;
}
```

Same type

real := integer

Both are strings

Type Checking Expressions

- The parser must perform type checking of every expression as part of its semantic actions.
- Add type checking to class **`ExpressionParser`** and to each statement parser.
- Flag type errors similarly to syntax errors.

Method **`ExpressionParser.parseTerm()`**

- Now besides doing syntax checking, our expression parser must also do type checking and determine the result type of each operation.

```
case PT_STAR:
{
    if (TypeChecker::are_both_integer(result_typespec,
                                      factor_typespec))
    {
        result_typespec = Predefined::integer_type;
    }

    else if (TypeChecker::is_at_least_one_real(result_typespec,
                                               factor_typespec))
    {
        result_typespec = Predefined::real_type;
    }

    else
    {
        error_handler.flag(expr_token, INCOMPATIBLE_TYPES, this);
    }

    break;
}
```

integer * integer ➔ integer result

one integer and one real, or both real ➔ real result

Type Checking Control Statements

- Method **`IfStatementParser.parse()`**

```
ICodeNode *IfStatementParser::parse_statement(Token *token) throw (string)
{
    token = next_token(token);  // consume the IF

    ICodeNode *if_node =
            ICodeFactory::create_icode_node((ICodeNodeType) NT_IF);

    Token *expr_token = new Token(*token);

    ExpressionParser expression_parser(this);
    ICodeNode *expr_node = expression_parser.parse_statement(token);
    if_node->add_child(expr_node);

    TypeSpec *expr_typespec = expr_node != nullptr
                                  ? expr_node->get_typespec()
                                  : Predefined::undefined_type;
    if (!TypeChecker::is_boolean(expr_typespec))
    {
        error_handler.flag(expr_token, INCOMPATIBLE_TYPES, this);
    }
    ...
}
```

ExpressionParser.parseFactor()

- Now an identifier can be more than just a variable name.

```
private ICodeNode parse_factor(Token token)
    throw (string)

{
    ...
    switch ((PascalTokenType) tokenType)
    {
        case IDENTIFIER:
        {
            return parse_identifier(token);
        }
    ...
}
```

ExpressionParser.parseIdentifier()

- Constant identifier

```
CONST
    pi = 3.14159;
```

  - Previously defined in a CONST definition.
  - Create an INTEGER_CONSTANT, REAL_CONSTANT, or a STRING_CONSTANT node.
  - Set its **VALUE** attribute.

```
TYPE
    direction =
        (north, south,
         east, west);
```

- Enumeration identifier
  - Previously defined in a type specification.
  - Create an INTEGER_CONSTANT node.
  - Set its **VALUE** attribute.

`ExpressionParser.parseIdentifier()`

- Variable identifier
  - Call method **`variableParser::parse()`**.