CMPE 152: Compiler Design

September 5 Class Meeting

Department of Computer Engineering
San Jose State University

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

SILICON VALLEY'S
FIRST CHOICE
FOR NEW
ENGINEERING HIRES
— Silicon Valley Business Journal, 2013

San José State
UNIVERSITY

Teams

Basic Scanning Algorithm

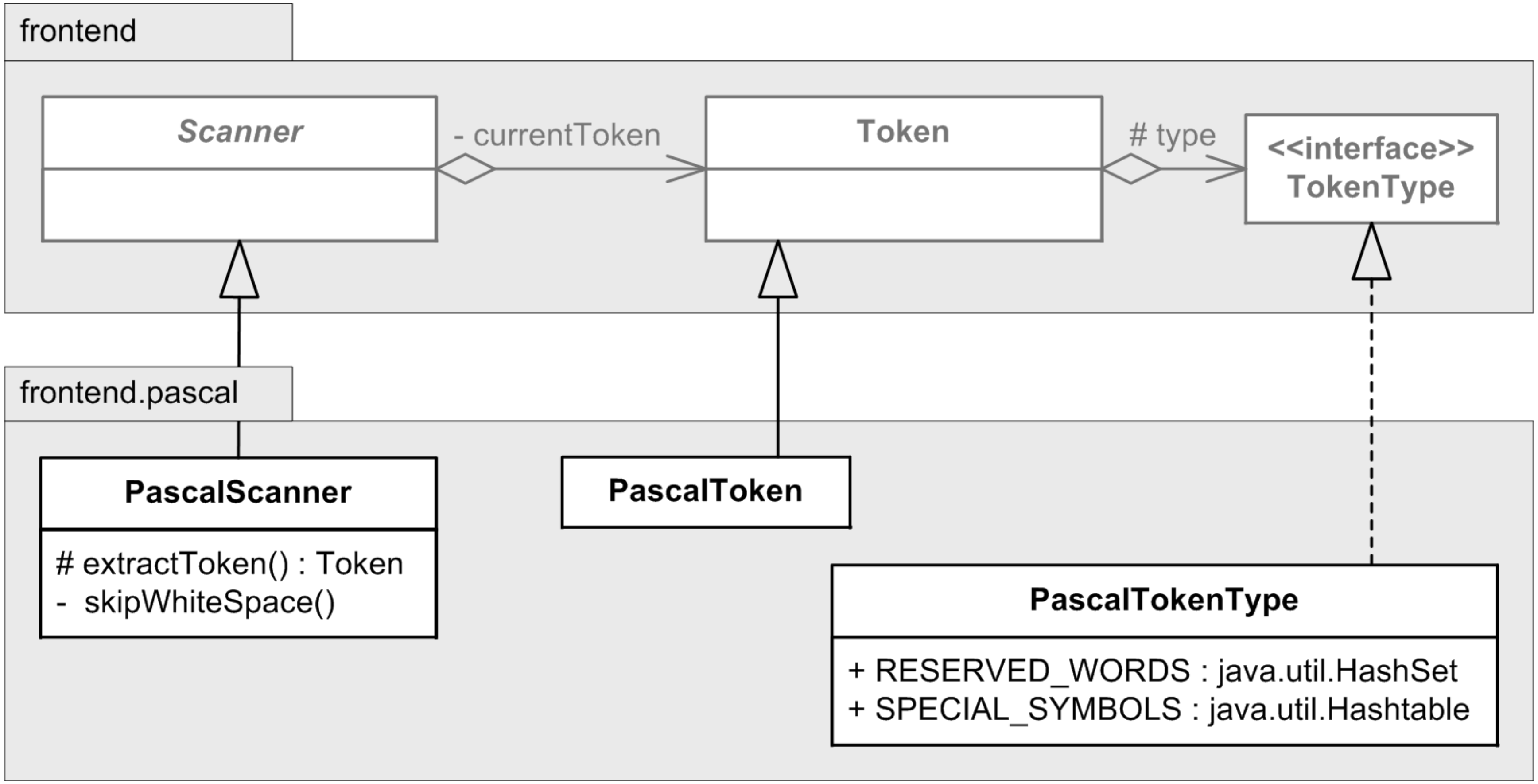
- Skip any blanks until the current character is nonblank.
  - In Pascal, a comment and the end-of-line character each should be treated as a blank.

- The current (nonblank) character determines what the next token is and becomes that token's first character.

- Extract the rest of the next token by copying successive characters up to but not including the first character that does not belong to that token.

- Extracting a token consumes all the source characters that constitute the token.
  - After extracting a token, the current character is the first character after the last character of that token.

Pascal-Specific Subclasses

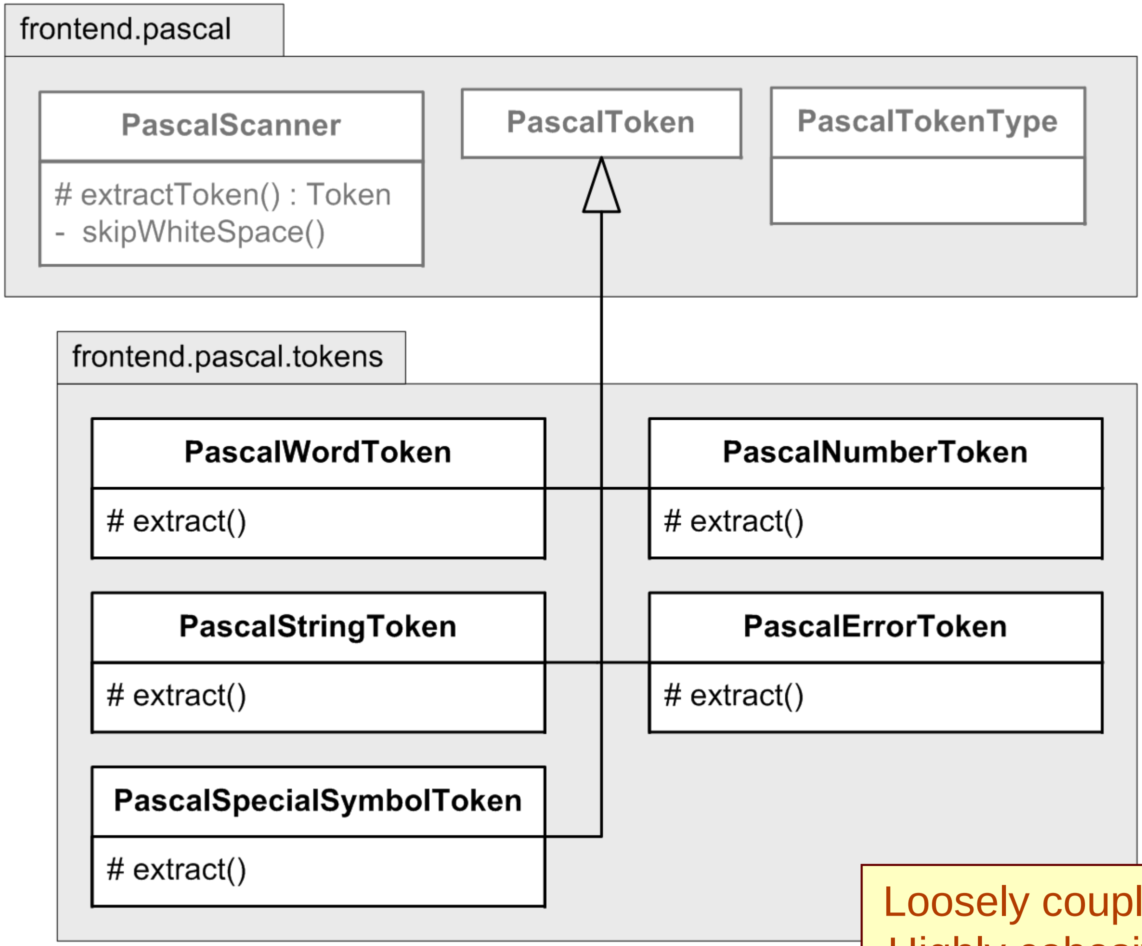


Class PascalScanner

```
Token *PascalScanner::extract_token() throw (string)
{
    skip_white_space();

    Token *token;
    char current_ch = current_char();
    string string_ch = " ";

    string_ch[0] = current_ch;

    // Construct the next token.  The current character determines the
    // token type.
    if (current_ch == Source::END_OF_FILE)
    {
        token = nullptr;
    }
    else if (isalpha(current_ch))
    {
        token = new PascalWordToken(source);
    }
    else if (isdigit(current_ch))
    {
        token = new PascalNumberToken(source);
    }
    ...
    return token;
}
```

The first character determines the type of the next token.

Pascal-Specific Token Classes



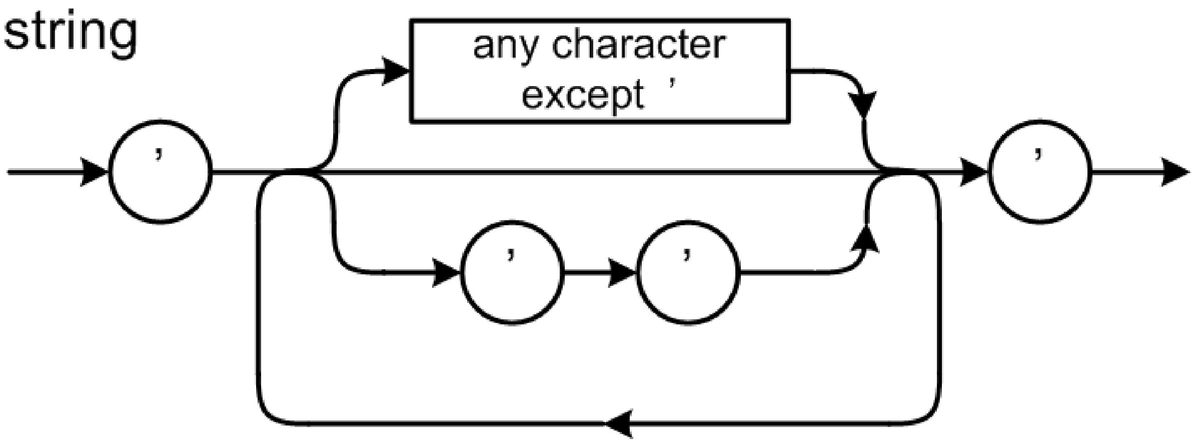
Loosely coupled.
Highly cohesive.

- Each class **PascalWordToken**, **PascalNumberToken**, **PascalStringToken**, **PascalSpecialSymbolToken**, and **PascalErrorToken** is is a subclass of class **PascalToken**.
  - **PascalToken** is a subclass of class **Token**.
- Each Pascal token subclass overrides the default **extract()** method of class **Token**.
  - The default method could only create single-character tokens.

Class PascalWordToken

```
void PascalWordToken::extract() throw (string)
{
    char current_ch = current_char();

    // Get the word characters (letter or digit). The scanner has
    // already determined that the first character is a letter.
    while (isalnum(current_ch))
    {
        text += current_ch;
        current_ch = next_char();  // consume character
    }

    // Is it a reserved word or an identifier?
    string upper_case(text);
    transform(upper_case.begin(), upper_case.end(),
              upper_case.begin(), ::toupper);
    if (PascalToken::RESERVED_WORDS.find(upper_case)
            != PascalToken::RESERVED_WORDS.end())
    {
        // Reserved word.
        type = (TokenType) PascalToken::RESERVED_WORDS[upper_case];
        value = new DataValue(upper_case);
    }
    else
    {
        // Identifier.
        type = (TokenType) PT_IDENTIFIER;
    }
}
```

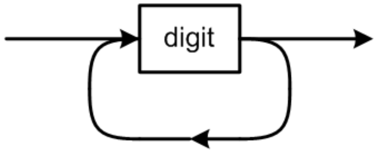
Pascal String Tokens



- A Pascal string literal constant uses *single* quotes.
- Two consecutive single quotes represents a single quote character inside a string.
  - `'Don''t'` is the string consisting of the characters `Don't`.
- A Pascal character literal constant is simply a string with only a single character, such as `'a'`.
- Pascal token subclass `PascalStringToken`.

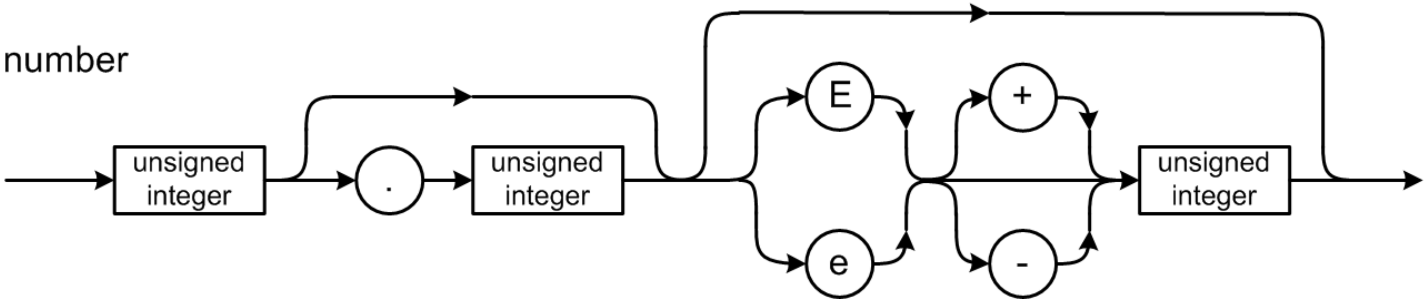
Pascal Number Tokens

unsigned integer



Any leading **+** or **–** sign before the literal constant is a separate token.

number



- A Pascal integer literal constant is an unsigned integer.
- A Pascal real literal constant starts with an unsigned integer (the whole part) followed by either
  - A decimal point followed by another unsigned integer (the fraction part), or
  - An **E** or **e**, optionally followed by **+** or **-**, followed by an unsigned integer (the exponent part), or
  - A whole part followed by an exponent part.

Class **PascalNumberToken**

- For the token string **"31415.926e-4"**, method **extractNumber()** passes the following parameter values to method **computeFloatValue()**:

wholeDigits	**"31415"**
fractionDigits	**"926"**
exponentDigits	**"4"**
exponentSign	**'-'**

- Compute variable **exponentValue**:

4	as computed by **computeIntegerValue()**
-4	after negation since **exponentSign** is '-'
-7	after subtracting **fractionDigits.length()**

- Compute **31415926 x 10^{-7} = 3.1415926**

A bit of a hack!

Syntax Error Handling

- Error handling is a three-step process:
  1. Detect the presence of a syntax error.
  2. Flag the error by pointing it out or highlighting it, and display a descriptive error message.
  3. Recover by moving past the error and resume parsing.
     - For now, we'll just move on, starting with the current character, and attempt to extract the next token.

- **SYNTAX_ERROR** message
  - source line number
  - beginning source position
  - token text
  - syntax error message

Class PascalParserTD

```
void PascalParserTD::parse() throw (string)
{
    ...

    // Loop over each token until the end of file.
    while ((token = next_token(token)) != nullptr)
    {
        TokenType token_type = token->get_type();
        last_line_number = token->get_line_number();

        string type_str;
        string value_str;

        switch ((PascalTokenType) token_type)
        {
            case PT_STRING:
            {
                type_str = "STRING";
                value_str = token->get_value()->s;
                break;
            }
```

Class **`PascalParserTD`**, *cont’d*

```
            case PT_IDENTIFIER:
            {
                type_str = "IDENTIFIER";
                value_str = "";
                break;
            }

            case PT_INTEGER:
            {
                type_str = "INTEGER";
                value_str = token->get_value()->display();
                break;
            }

            case PT_REAL:
            {
                type_str = "REAL";
                value_str = token->get_value()->display();
                break;
            }

            case PT_ERROR: break;
```

Class **`PascalParserTD`**, *cont'd*

```
    default:  // reserved word or special character
    {
        DataValue *token_value = token->get_value();

        // Reserved word
        if (token_value != nullptr)
        {
            value_str = token_value->s;
            type_str = value_str;
        }

        // Special symbol
        else
        {
            type_str =
                PascalToken::SPECIAL_SYMBOL_NAMES[
                                   (PascalTokenType) token_type];
        }

        break;
    }
}
```

Class **PascalParserTD**, *cont’d*

```
        if (token_type != (TokenType) PT_ERROR)
        {
            // Format and send a message about each token.
            Message message(TOKEN,
                            LINE_NUMBER, to_string(token->get_line_number()),
                            POSITION, to_string(token->get_position()),
                            TOKEN_TYPE, type_str,
                            TOKEN_TEXT, token->get_text(),
                            TOKEN_VALUE, value_str);
            send_message(message);
        }
        else
        {
            PascalErrorCode error_code =
                                (PascalErrorCode) token->get_value()->i;
            error_handler.flag(token, error_code, this);
        }
    }

    ...
}
```

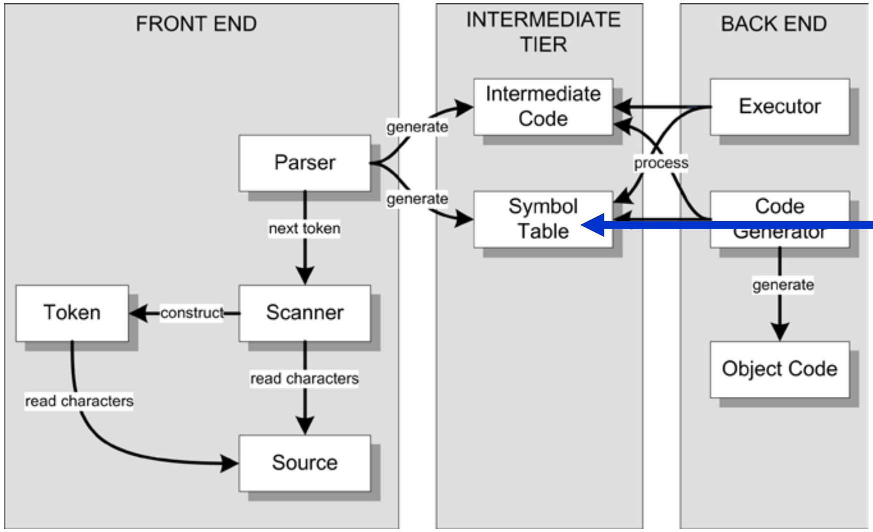
Program: Pascal Tokenizer

- Verify the correctness of the Pascal token subclasses.
- Verify the correctness of the Pascal scanner.

- Demo (Chapter 3)

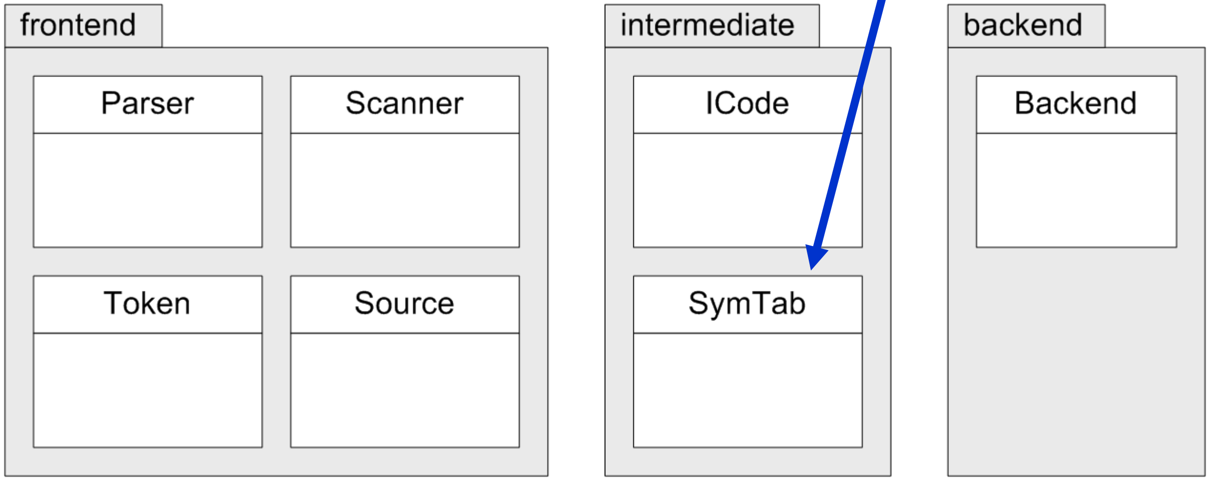
Quick Review of the Framework

FROM:



Our next topic:
The **symbol table**

TO:



frontend

Parser

Scanner

Token

Source

intermediate

ICode

SymTab

backend

Backend

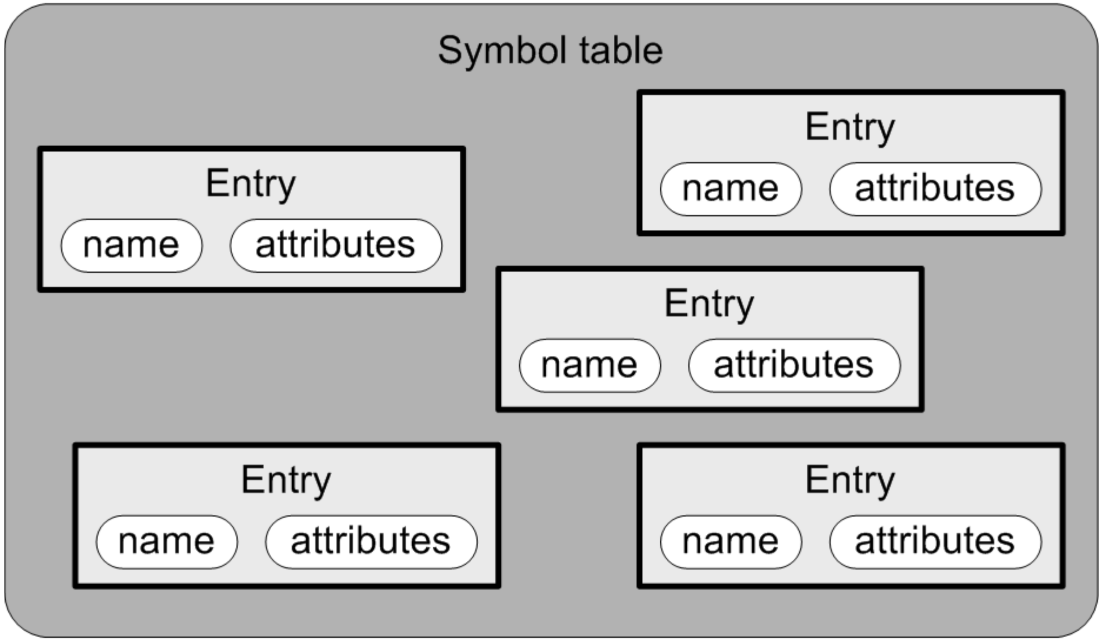
The Symbol Table: Basic Concepts

- Purpose
  - To store information about certain tokens during the translation process (i.e., parsing and scanning)
- What information to store?
  - Anything that’s useful!
  - For an identifier:
    - name
    - data type
    - how it’s defined (as a variable, type, function name, etc.)

The Symbol Table: Basic Operations

- Enter new information.
- Look up existing information.
- Update existing information.

The Symbol Table: Conceptual Design



Goal: The symbol table should be source language independent.

- Each entry in the symbol table has
  - a name
  - attributes
- At the conceptual level, we don't worry about implementation.

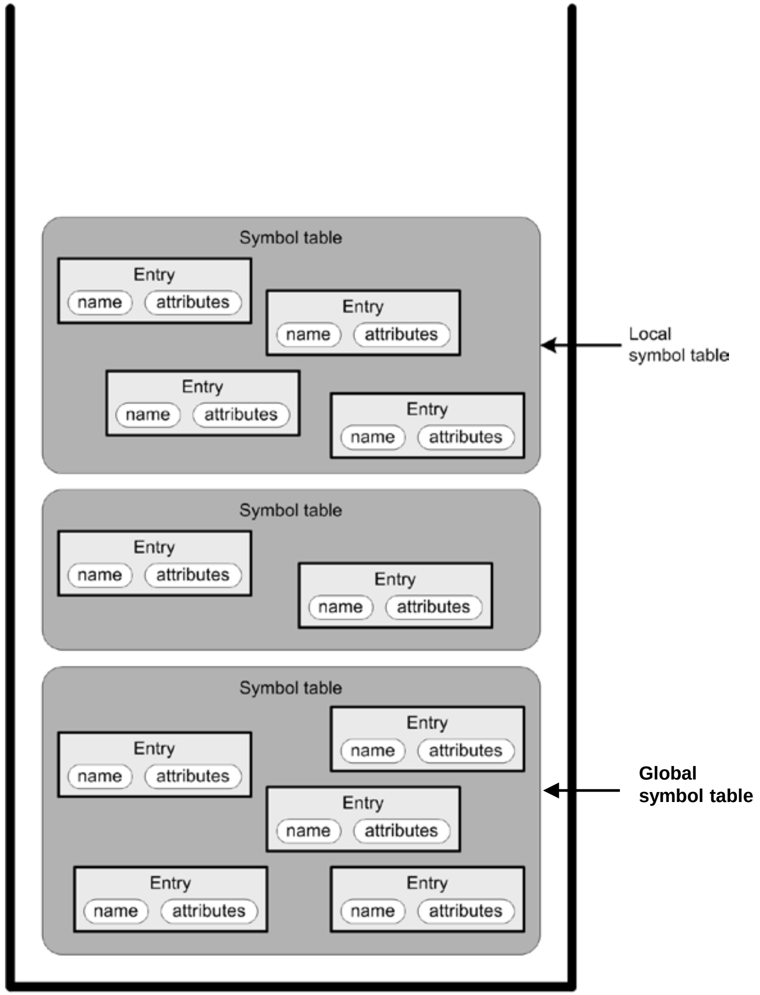
What Needs a Symbol Table?

- A Pascal program
  - Identifiers for constant, type, variable, procedure, and function names.
- A Pascal procedure or function
  - Identifiers for constant, type, variable, procedure, and function names.
  - Identifiers for formal parameter (argument) names.
- A Pascal record type
  - Identifiers for field names.

The Symbol Table Stack

- Language constructs can be nested.
  - Procedures and functions are nested inside a program.
  - Procedures and functions can be nested inside of each other.
  - Record types are defined within programs, procedures, and functions.
  - Record types can be nested inside of each other.
- Therefore, symbol tables need to be kept on a symbol table stack.

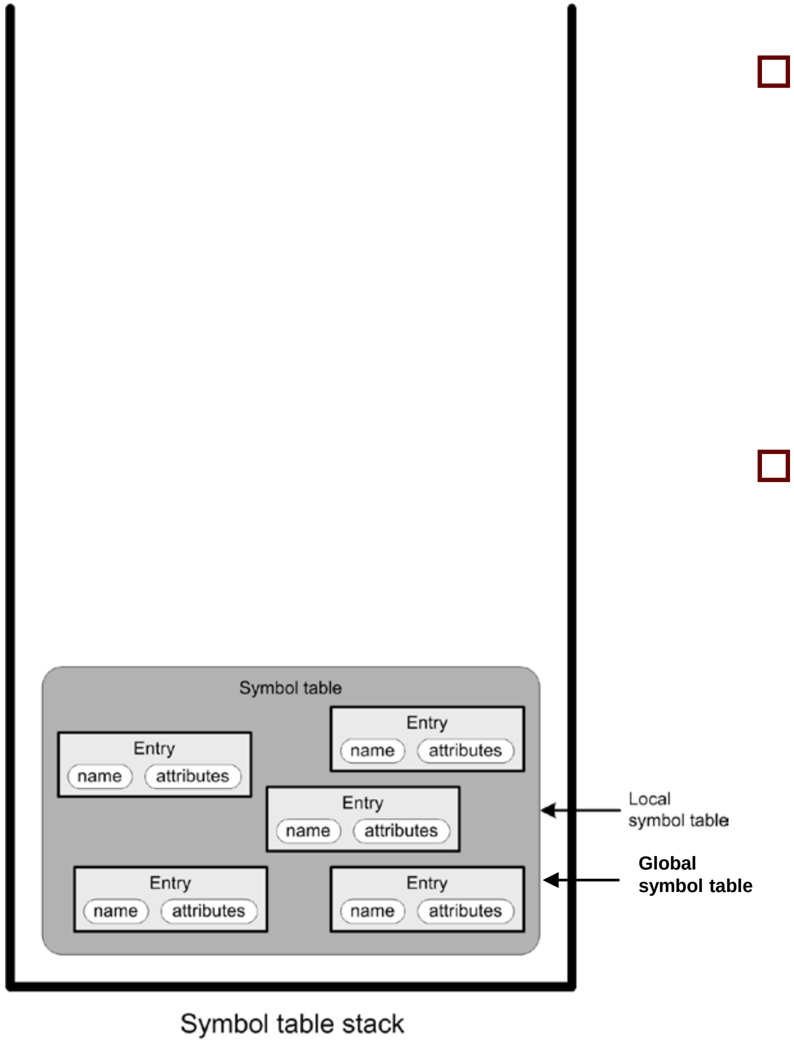
The Symbol Table Stack, *cont’d*



Symbol table stack

- Whichever symbol table is on top of the stack is the local symbol table.
- The first symbol table created (the one at the bottom of the stack) is the global symbol table.
  - It stores the predefined information, such as entries for the names of the standard types `integer`, `real`, `char`, and `boolean`.
- During the translation process, symbol tables are pushed onto and popped off the stack …
  - … as the parser enters and exits nested procedures, functions, record types, etc.

The Symbol Table Stack, *cont'd*



- For now, we'll have only have a single symbol table.
  - Therefore, the local symbol table is the global symbol table.
- We won't need multiple symbol tables until we start to parse declarations.
  - Implementing the symbol table stack now will make things easier for us later.