CS 153: Concepts of Compiler Design

September 5 Class Meeting

Department of Computer Science
San Jose State University

Fall 2017

Instructor: Ron Mak

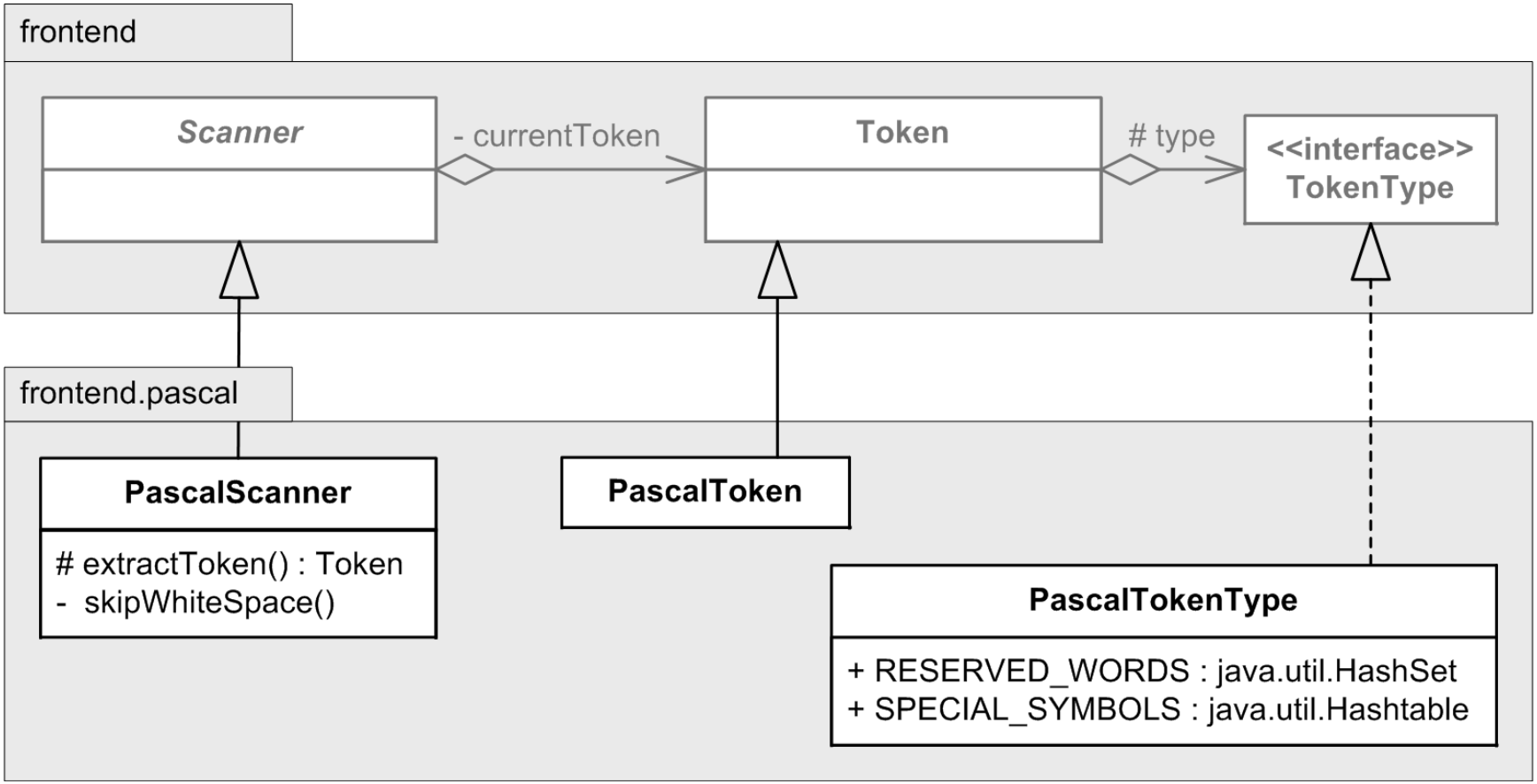
www.cs.sjsu.edu/~mak

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

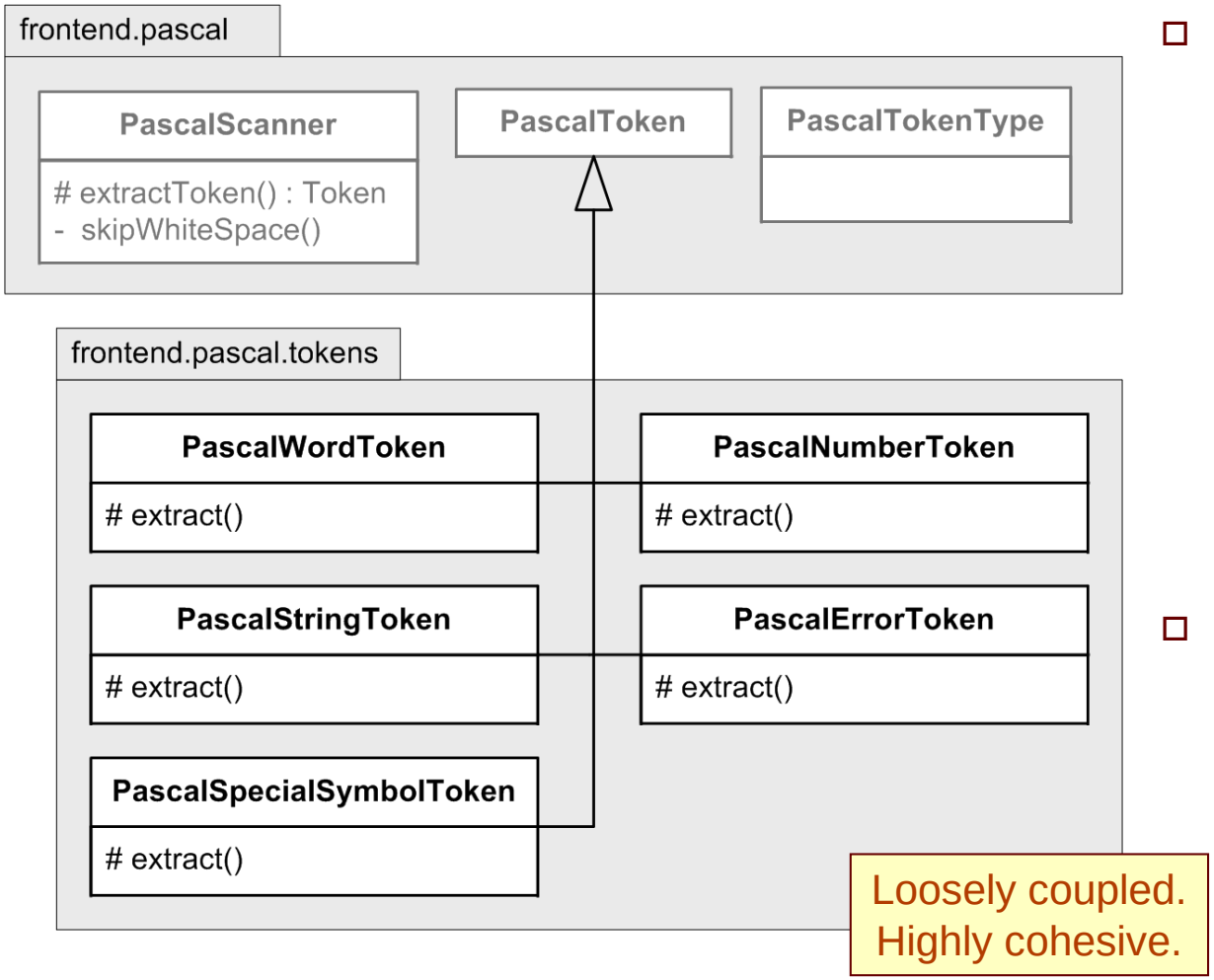


Class **PascalScanner**

```
protected Token extractToken()
    throws Exception
{
    skipWhiteSpace();

    Token token;
    char currentChar = currentChar();

    // Construct the next token. The current character determines the
    // token type.
    if (currentChar == EOF) {
        token = new EofToken(source);
    }
    else if (Character.isLetter(currentChar)) {
        token = new PascalWordToken(source);
    }
    else if (Character.isDigit(currentChar)) {
        token = new PascalNumberToken(source);
    }
    ...
    return token;
}
```

The first character determines the type of the next token.

Pascal-Specific Token Classes

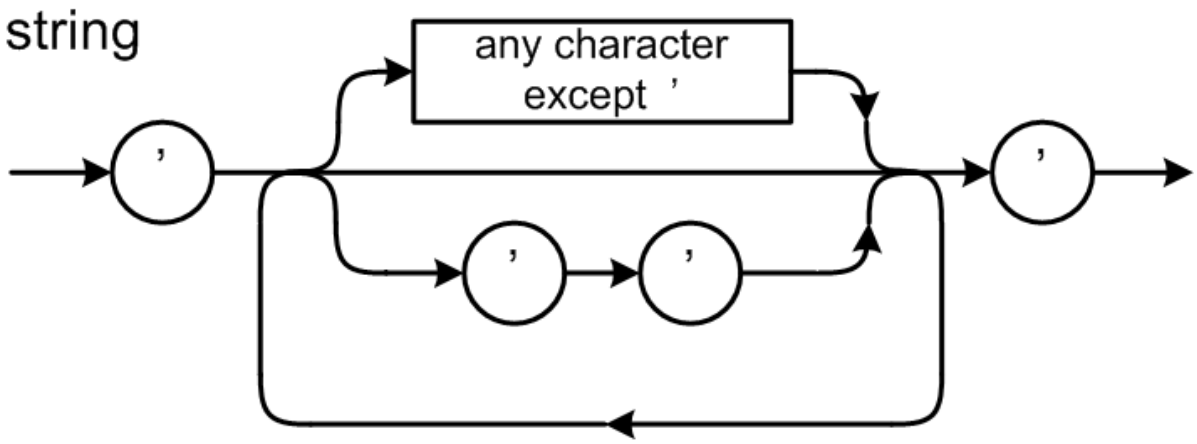


- Each class **`PascalWordToken`**, **`PascalNumberToken`**, **`PascalStringToken`**, **`PascalSpecial-SymbolToken`**, and **`PascalErrorToken`** is is a subclass of class **`PascalToken`**.
  - **`PascalToken`** is a subclass of class **`Token`**.
- Each Pascal token subclass overrides the default **`extract()`**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

Class **PascalWordToken**

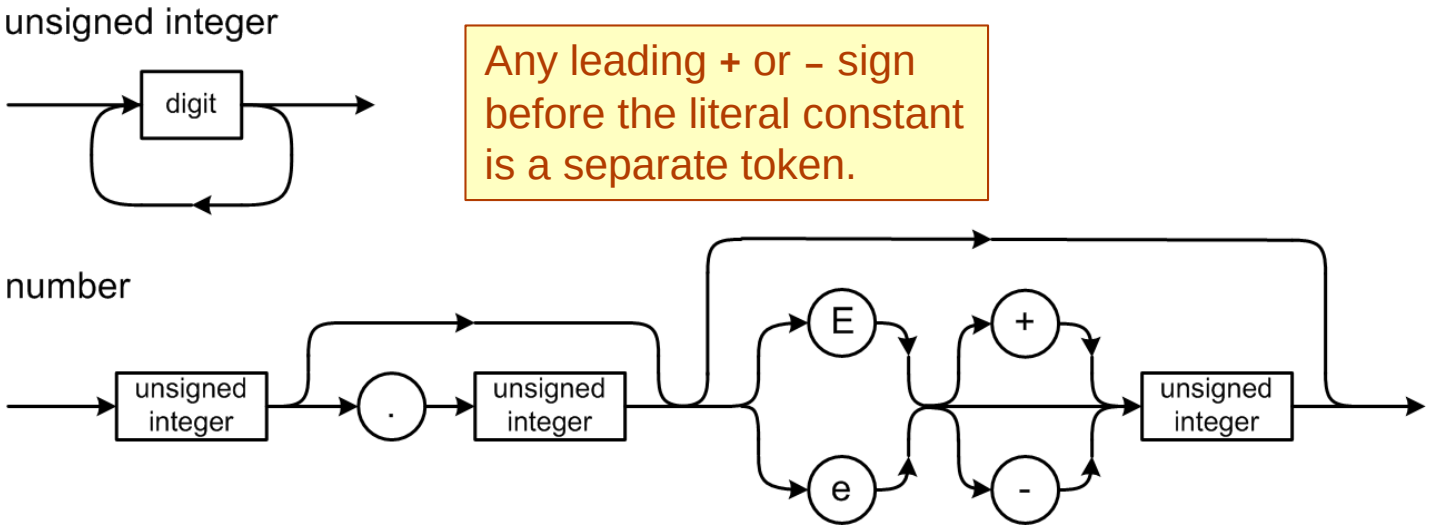
```
protected void extract()
    throws Exception
{
    StringBuilder textBuffer = new StringBuilder();
    char currentChar = currentChar();

    // Get the word characters (letter or digit).  The scanner has
    // already determined that the first character is a letter.
    while (Character.isLetterOrDigit(currentChar)) {
        textBuffer.append(currentChar);
        currentChar = nextChar();  // consume character
    }

    text = textBuffer.toString();

    // Is it a reserved word or an identifier?
    type = (RESERVED_WORDS.contains(text.toLowerCase()))
           ? PascalTokenType.valueOf(text.toUpperCase())  // reserved word
           : IDENTIFIER;                                  // 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



- 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 $31415926 \times 10^{-7} = 3.1415926$

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**

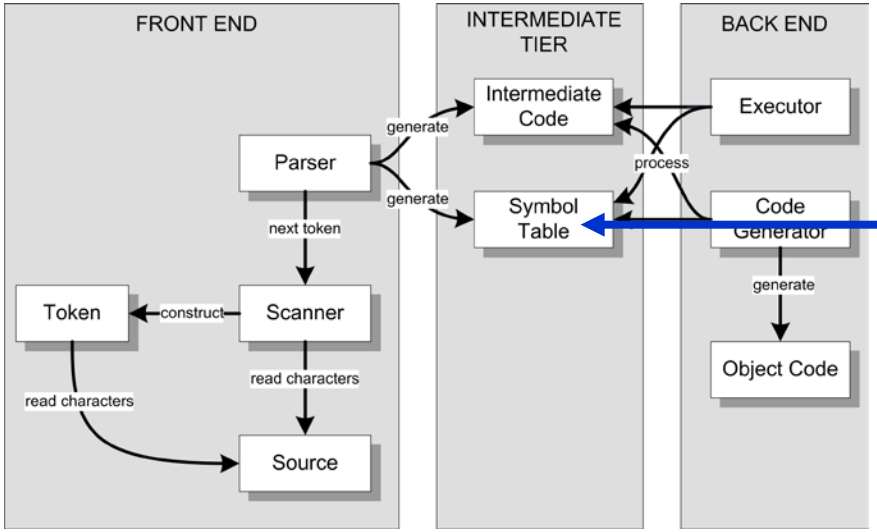
```
public void parse()
    throws Exception
{
    ...
        // Loop over each token until the end of file.
        while (!((token = nextToken()) instanceof EofToken)) {
            TokenType tokenType = token.getType();

            if (tokenType != ERROR) {

                // Format each token.
                sendMessage(new Message(TOKEN,
                                        new Object[] {token.getLineNumber(),
                                                      token.getPosition(),
                                                      tokenType,
                                                      token.getText(),
                                                      token.getValue()}));
            }
            else {
                errorHandler.flag(token, (PascalErrorCode) token.getValue(), this);
            }
        }
    ...
}
```

Program: Pascal Tokenizer

- Verify the correctness of the Pascal token subclasses.
- Verify the correctness of the Pascal scanner.
- Demo (Chapter 3)

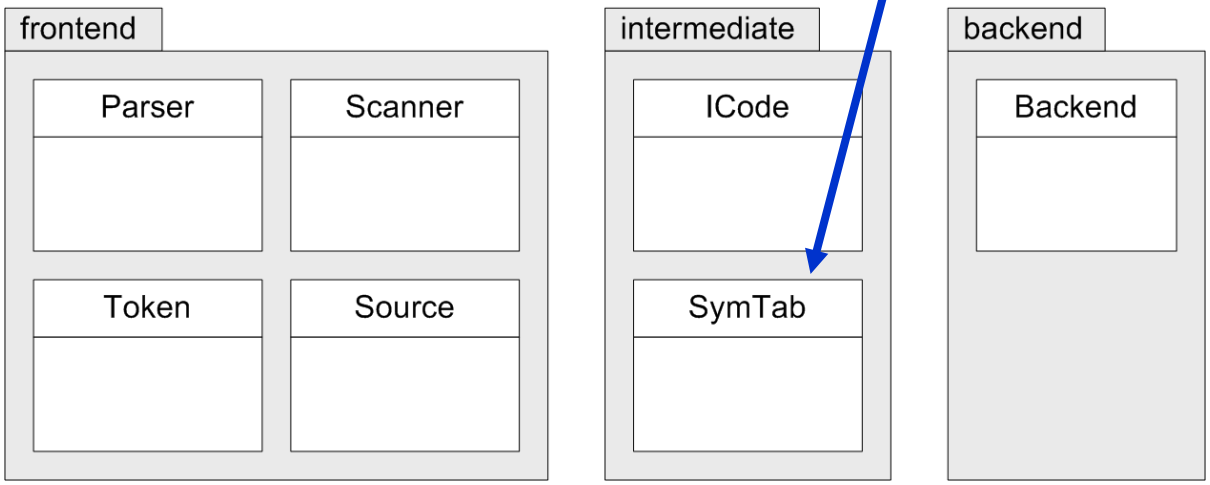
Quick Review of the Framework

FROM:



Our next topic:
The **symbol table**

TO:



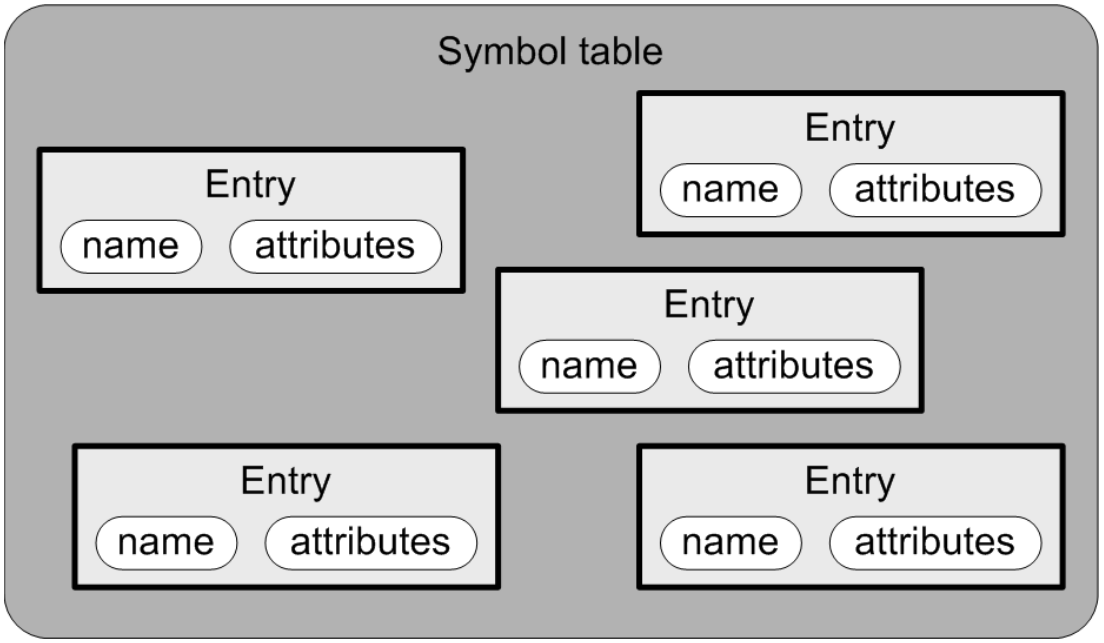
The Symbol Table: Basic Concepts

- 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.

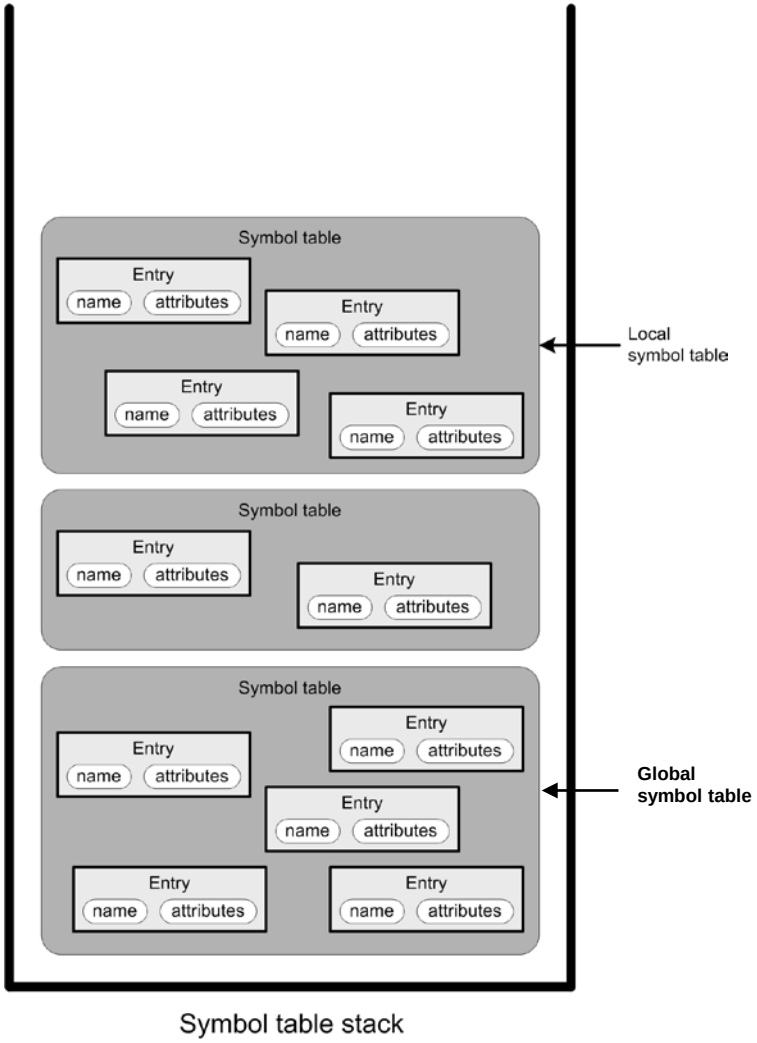
What Needs a Symbol Table?

- 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.

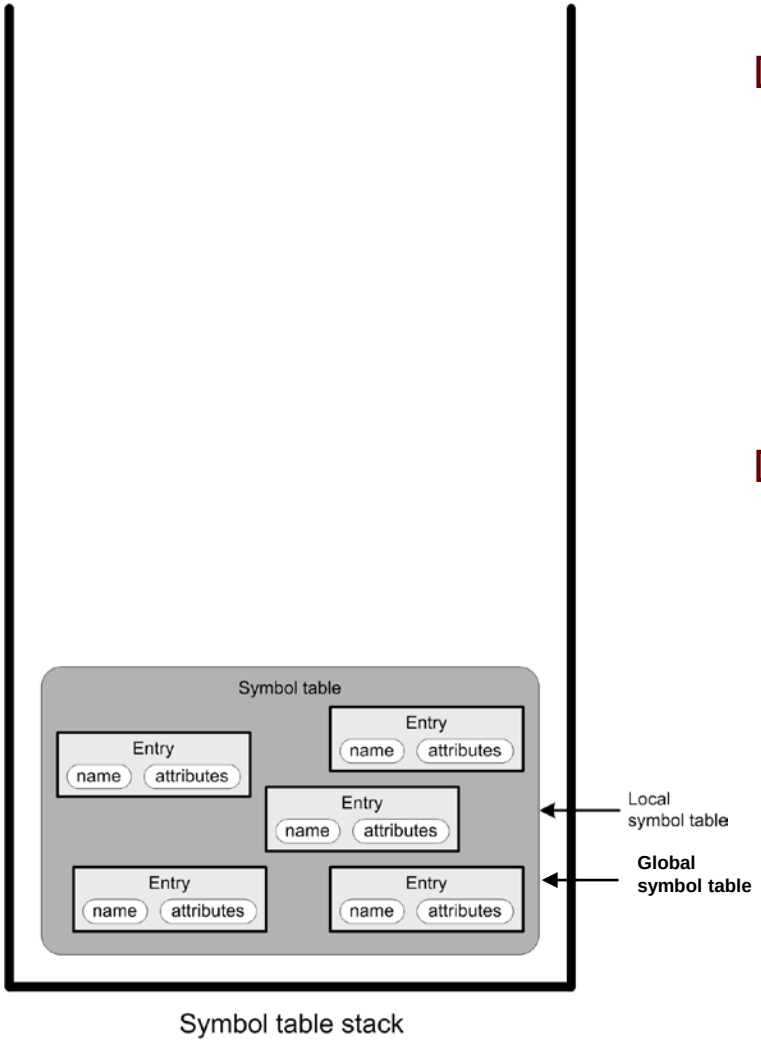
The Symbol Table Stack, *cont'd*



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.

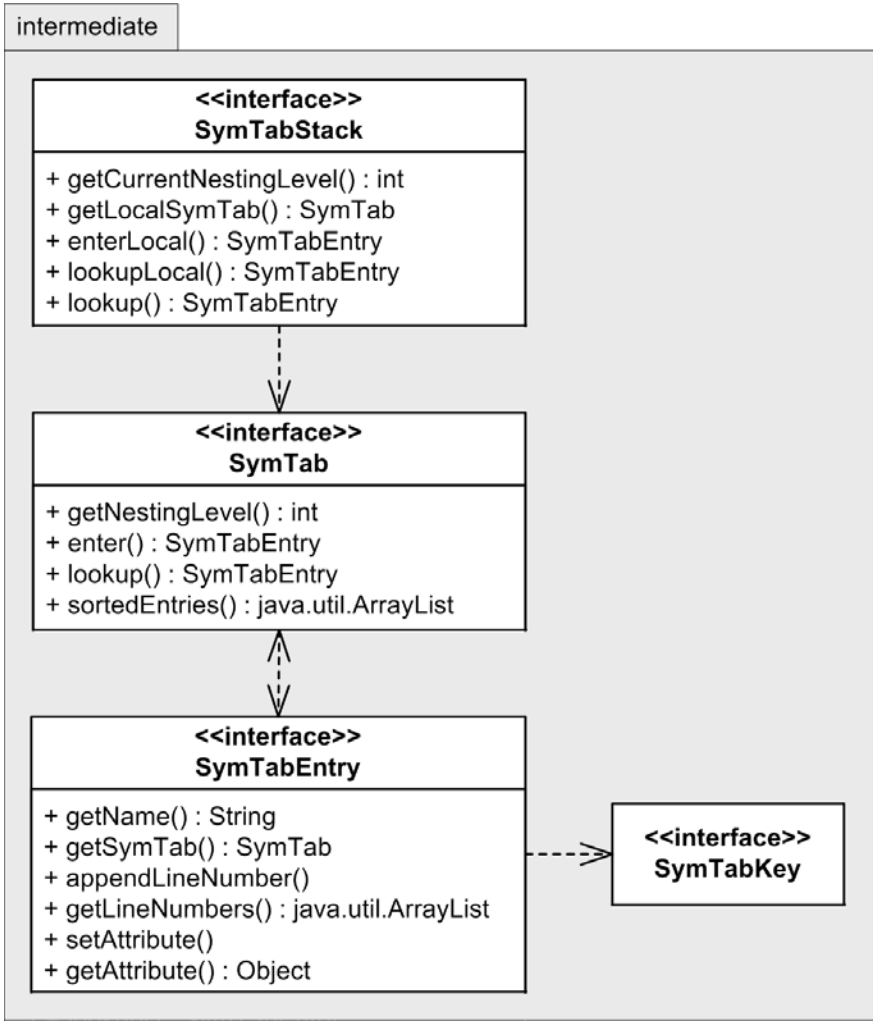
The Symbol Table Stack, *cont'd*



Symbol table stack

- 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.

Symbol Table Interfaces



- Key operations
  - Enter into the local symbol table, the table currently at the top of the stack.
  - Look up (search for) an entry only in the local symbol table.
  - Look up an entry in all the symbol tables in the stack.
    - Search from the top (the local) table down to the bottom (global) table.
- Each symbol table has a nesting level.
  - 0: global
  - 1: program
  - 2: top-level procedure
  - 3: nested procedure, etc.

Symbol Table Interfaces, *cont'd*

- Java interfaces
  - Package **wci.intermediate**
    - **SymTabStack**
    - **SymTab**
    - **SymTabEntry**
    - **SymTabKey**
  - Example:

```
public interface SymTabEntry
{
    public String getName();
    public SymTab getSymTab();
    public void setAttribute(SymTabKey key, Object value);
    public Object getAttribute(SymTabKey key);
    public void appendLineNumber(int lineNumber);
    public ArrayList<Integer> getLineNumbers();
}
```

For cross-referencing.

Why All the Interfaces?

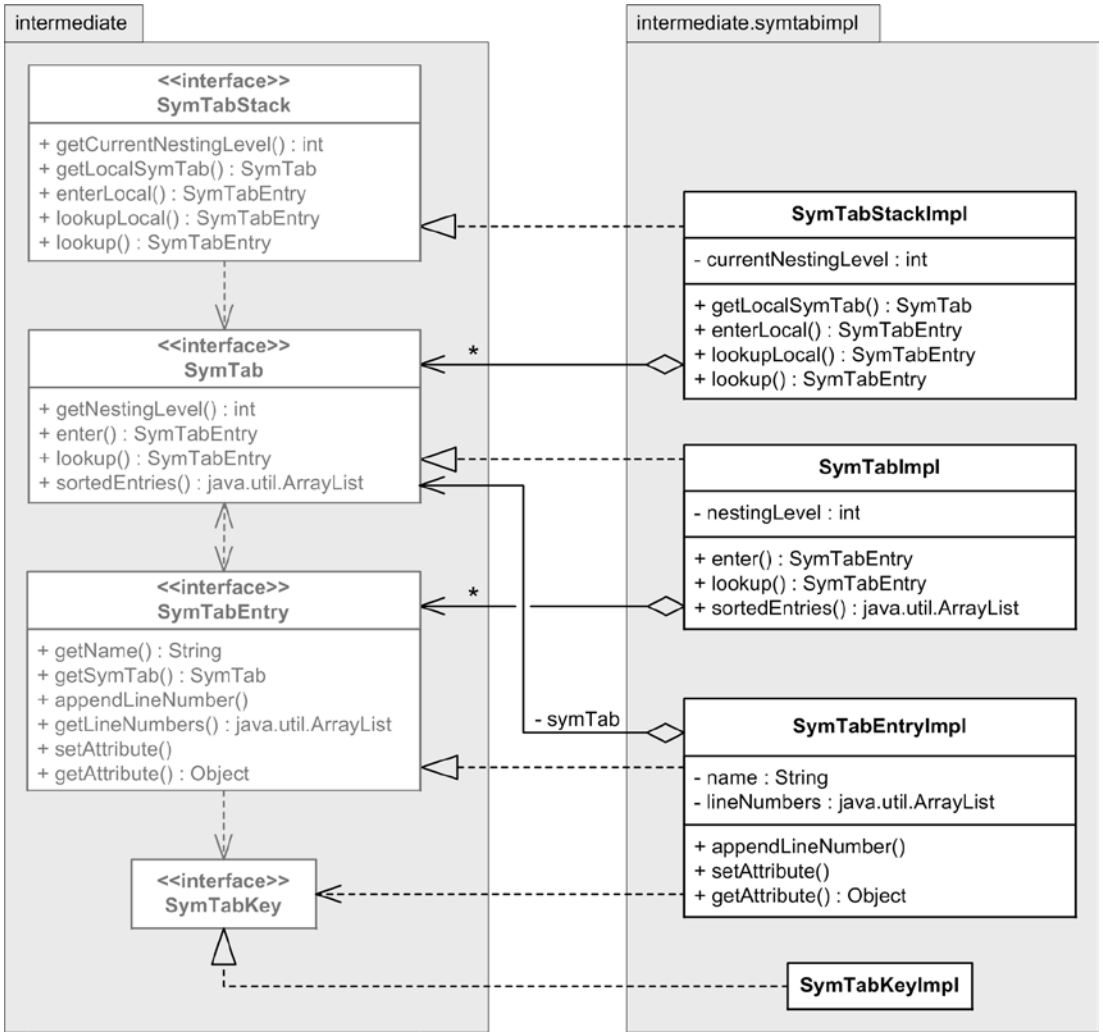
- We've defined the symbol table components entirely with interfaces.
- Other components that use the symbol table will code to the interfaces, not to specific implementations.
  - Loose coupling provides maximum support for flexibility.

```
symTabStack = SymTabFactory.createSymTabStack();
SymTabEntry entry = symTabStack.lookup(name);
```

Why All the Interfaces? *cont'd*

- We'll be able to implement the symbol table however we like.
- We can change the implementation in the future without affecting the users.
  - But not change the interfaces.
- The interfaces provide an API for the symbol table.
  - Callers of the symbol table API only need to understand the symbol table at the conceptual level.

Symbol Table Components Implementation



- Implementation classes are defined in package **`intermediate.symtabimpl`**
- A **`SymTabStackImpl`** object can own zero or more **`SymTab`** objects.
- A **`SymTabImpl`** object can own zero or more **`SymTabEntry`** objects.
- A **`SymTabEntryImpl`** object maintains a reference to the **`SymTab`** object that contains it.

–

A Symbol Table Factory Class

```
public class SymTabFactory
{
    public static SymTabStack createSymTabStack()
    {
        return new SymTabStackImpl();
    }

    public static SymTab createSymTab(int nestingLevel)
    {
        return new SymTabImpl(nestingLevel);
    }

    public static SymTabEntry createSymTabEntry(String name, SymTab symTab)
    {
        return new SymTabEntryImpl(name, symTab);
    }
}
```

Symbol Table Implementation

- Implement the symbol table as a Java hash table.
  - **Key:** the identifier name (a string)
  - **Value:** the symbol table entry corresponding to the name
- Even better:

```
extends TreeMap<String, SymTabEntry>
implements SymTab
```

  - Like a hash table except that it keeps its entries sorted.

```
public SymTabEntry enter(String name)
{
    SymTabEntry entry = SymTabFactory.createSymTabEntry(name, this);
    put(name, entry);

    return entry;
}

public SymTabEntry lookup(String name)
{
    return get(name);
}
```

Symbol Table Implementation, *cont'd*

- Method **`sortedEntries()`** returns an array list of the symbol table entries in sorted order.

```
public ArrayList<SymTabEntry> sortedEntries()
{
    Collection<SymTabEntry> entries = values();
    Iterator<SymTabEntry> iter = entries.iterator();
    ArrayList<SymTabEntry> list = new ArrayList<SymTabEntry>(size());

    // Iterate over the sorted entries and append them to the list.
    while (iter.hasNext()) {
        list.add(iter.next());
    }

    return list;  // sorted list of entries
}
```

Symbol Table Stack Implementation

- Implement the stack as an array list of symbol tables:

```
extends ArrayList<SymTab>
implements SymTabStack
```

- Constructor
  - For now, the current nesting level will always be 0.
  - Initialize the stack with the global symbol table.
    - For now, that's the only symbol table, so it's also the local table.

Who calls this constructor?

```
public SymTabStackImpl()
{
    this.currentNestingLevel = 0;
    add(SymTabFactory.createSymTab(currentNestingLevel));
}
```

Symbol Table Stack Implementation, *cont'd*

```
public SymTabEntry enterLocal(String name)
{
    return get(currentNestingLevel).enter(name);
}

public SymTabEntry lookupLocal(String name)
{
    return get(currentNestingLevel).lookup(name);
}

public SymTabEntry lookup(String name)
{
    return lookupLocal(name);
}
```

- For now, since there is only one symbol table on the stack, method **`lookup()`** simply calls method **`lookupLocal()`**.
  - In the future, method **`lookup()`** will search the entire stack.
  - Why do we need both methods?

Assignment #2

- Write a scanner for the Java language.
- Add a new Java front end.