CMPE 152: Compiler Design

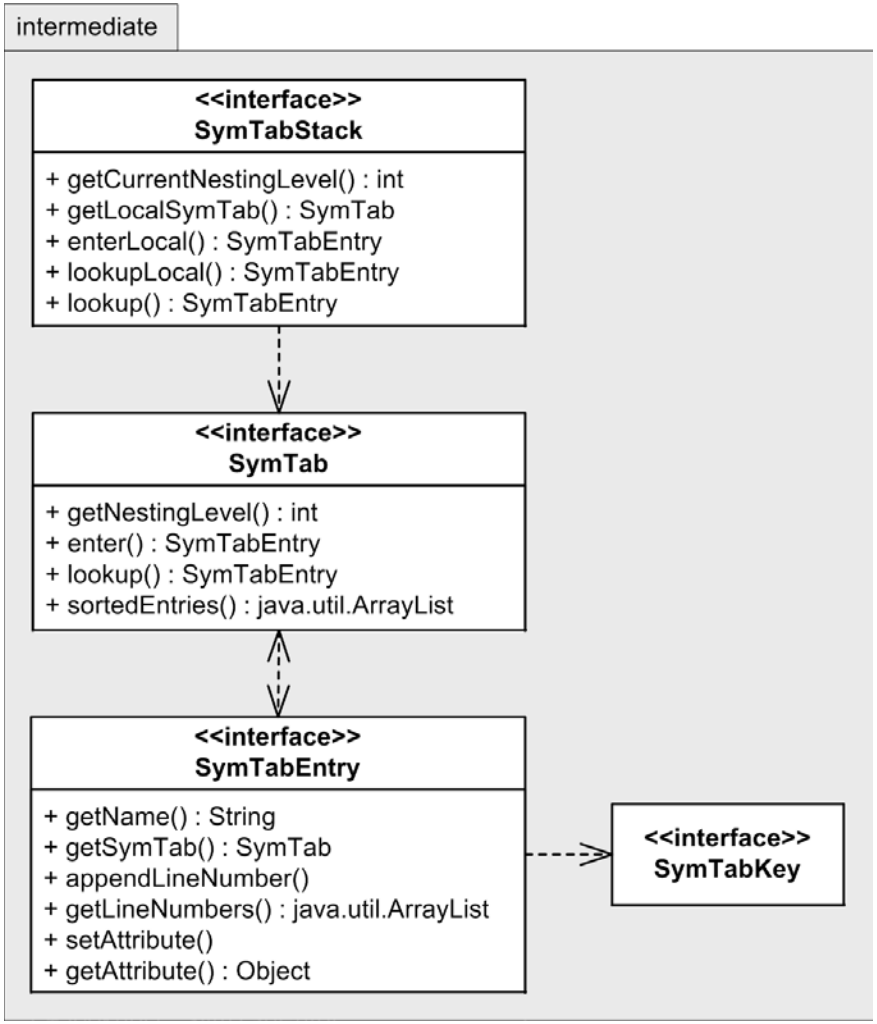
September 5 Lab

Department of Computer Engineering
San Jose State University

Fall 2017
Instructor: Ron Mak

www.cs.sjsu.edu/~mak

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*

- Abstract classes as interfaces
  - Namespace **`wci::intermediate`**
  - **`SymTabStack`**
  - **`SymTab`**
  - **`SymTabEntry`**
  - **`SymTabKey`**

Symbol Table Interfaces, *cont'd*

```
/**
 * Symbol table entry keys.
 */
enum class SymTabKey
{
    // to be "subclassed" by implementation-specific symbol table keys
};

class SymTab;    // forward declaration

/**
 * Symbol table entry value.
 */
struct EntryValue
{
    DataValue *value;
    SymTab    *symtab;

    EntryValue()                 : value(nullptr), symtab(nullptr) {};
    EntryValue(DataValue *value) : value(value), symtab(nullptr) {};

    ~EntryValue() {}
};
```

Symbol Table Interfaces, *cont'd*

```
class SymTabEntry
{
public:
    SymTabEntry(const string name, SymTab *symtab);
    virtual ~SymTabEntry();
    virtual string get_name() const = 0;
    virtual SymTab *get_symtab() const = 0;
    virtual void set_attribute(const SymTabKey key, EntryValue *value) = 0;
    virtual EntryValue *get_attribute(const SymTabKey key) = 0;

    virtual void append_line_number(const int line_number) = 0;
    virtual vector<int> get_line_numbers() = 0;
};
```

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.

```
symtab_stack = SymTabFactory::create_symtab_stack();
SymTabEntry *entry = symtab_stack->lookup(name);
```

Why All the Interfaces? *cont'd*

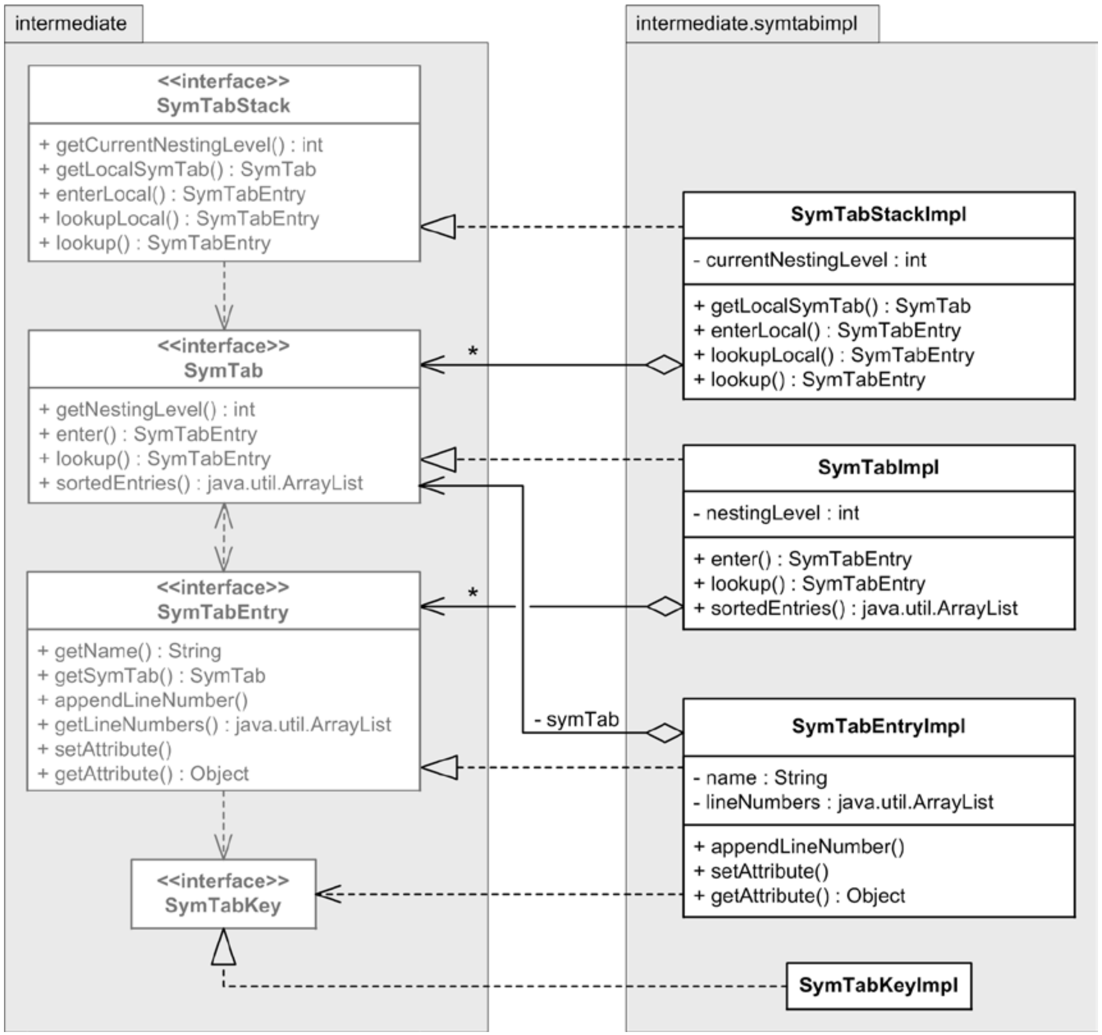
- We'll be able to implement the symbol table however we like.

- 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 <u>conceptual level</u>.

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

```
SymTabStack *SymTabFactory::create_symtab_stack()
{
    return new SymTabStackImpl();
}

SymTab *SymTabFactory::create_symtab(int nesting_level)
{
    return new SymTabImpl(nesting_level);
}

SymTabEntry *SymTabFactory::create_symtab_entry(string name, SymTab *symtab)
{
    return new SymTabEntryImpl(name, symtab);
}
```

Symbol Table Implementation

- Implement the symbol table as a map.
  - **Key:** the identifier name (a string)
  - **Value:** pointer to the symbol table entry corresponding to the name

SymTabImpl.h

```
private:
    map<string, SymTabEntryImpl *> contents;
```

Symbol Table Implementation

```
SymTabEntry *SymTabImpl::enter(string name)
{
    SymTabEntry *entry = SymTabFactory::create_symtab_entry(name, this);
    contents[name] = (SymTabEntryImpl *) entry;

    return entry;
}

SymTabEntry *SymTabImpl::lookup(const string name)
{
    return (contents.find(name) != contents.end())
                ? contents[name]
                : nullptr;
}
```

Symbol Table Implementation, *cont'd*

- Member function **sorted_entries()** returns an array list of the symbol table entries in sorted order.

```
vector<SymTabEntry *> SymTabImpl::sorted_entries()
{
    vector<SymTabEntry *> list;
    map<string, SymTabEntryImpl *>::iterator it;

    for (it = contents.begin(); it != contents.end(); it++)
    {
        list.push_back(it->second);
    }

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

Symbol Table Stack Implementation

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

SymTabStackImpl.h

```
private:
    vector<SymTab *> stack;
```

- 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.

```
SymTabStackImpl::SymTabStackImpl()
{
    stack.push_back(SymTabFactory::create_symtab(0));
}
```

Who calls this constructor?

Symbol Table Stack Implementation, *cont'd*

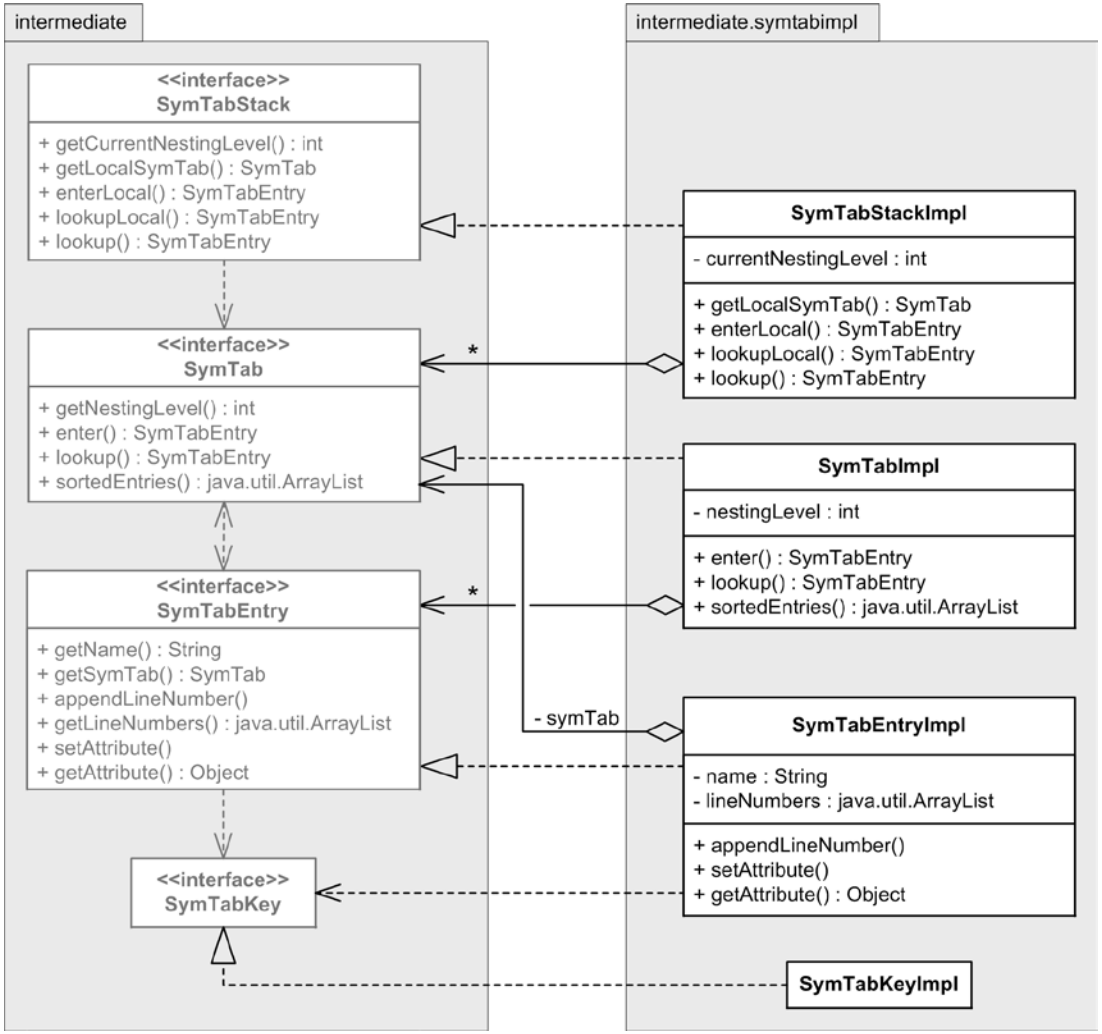
```
SymTabEntry *SymTabStackImpl::enter_local(const string name)
{
    return stack[current_nesting_level]->enter(name);
}

SymTabEntry *SymTabStackImpl::lookup_local(const string name) const
{
    return stack[current_nesting_level]->lookup(name);
}

SymTabEntry *SymTabStackImpl::lookup(const string name) const
{
    return lookup_local(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?

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.

Symbol Table Entry Implementation

```
enum class SymTabKeyImpl
{
    // Constant.
    CONSTANT_VALUE,

    // Procedure or function.
    ROUTINE_CODE, ROUTINE_SYMTAB, ROUTINE_ICODE,
    ROUTINE_PARMS, ROUTINE_ROUTINES,

    // Variable or record field value.
    DATA_VALUE
};
```

```
struct EntryValue
{
    DataValue *value;
    SymTab    *symtab;

    EntryValue()                  : value(nullptr), symtab(nullptr) {};
    EntryValue(DataValue *value) : value(value), symtab(nullptr) {};

    ~EntryValue() {}
};
```

Symbol Table Entry Implementation

- Implement a symbol table entry as a Java hash map:

```
private:
    map<SymTabKey, EntryValue *> contents;
```

- Most flexibility in what we can store in each entry.
  - **Key:** attribute key (an enumerated type)
  - **Value:** pointer to an `EntryValue` object

```
void SymTabEntryImpl::set_attribute(const SymTabKey key, EntryValue *value)
{
    contents[key] = value;
}

EntryValue *SymTabEntryImpl::get_attribute(const SymTabKey key)
{
    return contents.find(key) != contents.end() ? contents[key]
                                                : nullptr;
}
```

What to Store in Each Symbol Table Entry

- Each symbol table entry is designed to store information about an identifier.
- The attribute keys indicate what information we will store for each type of identifier.
- Store common information in fixed fields (e.g., **`lineNumbers`**) and store identifier type-specific information as attributes values.

Modifications to Class **PascalParserTD**

```
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;

    // Cross reference only the identifiers.
    if (token_type == (TokenType) PT_IDENTIFIER)
    {
        string name = token->get_text();
        transform(name.begin(), name.end(), name.begin(), ::tolower);

        // If it's not already in the symbol table,
        // create and enter a new entry for the identifier.
        SymTabEntry *entry = symtab_stack->lookup(name);
        if (entry == nullptr) entry = symtab_stack->enter_local(name);

        // Append the current line number to the entry.
        entry->append_line_number(token->get_line_number());
    }
```

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

```
        else if (token_type == (TokenType) PT_ERROR)
        {
            PascalErrorCode error_code =
                                    (PascalErrorCode) token->get_value()->i;
            error_handler.flag(token, error_code, this);
        }
    }
```

Cross-Reference Listing

- A cross-reference listing verifies the symbol table code:

```
java -classpath classes Pascal compile -x newton.pas
```

- Modifications to the main **Pascal** class:

```
bool xref = flags.find('x') != string::npos;
...
parser->parse();
...
symtab_stack = parser->get_symtab_stack();
icode = parser->get_icode();

if (xref)
{
    CrossReferencer cross_referencer;
    cross_referencer.print(symtab_stack);
}
```

New utility class **CrossReferencer** generates the cross-reference listing.

Demo

Cross-Reference Listing Output

```
001 PROGRAM newton (input, output);
002
003 CONST
004     EPSILON = 1e-6;
005
006 VAR
007     number       : integer;
008     root, sqRoot : real;
009
010 BEGIN
011     REPEAT
012         writeln;
013         write('Enter new number (0 to quit): ');
014         read(number);

            ...

035     UNTIL number = 0
036 END.

                  36 source statements.
                   0 syntax errors.
                0.06 seconds total parsing time.
```

Cross-Reference Listing Output, *cont'd*

```
===== CROSS-REFERENCE TABLE =====

Identifier         Line numbers
----------         ------------
abs                031 033
epsilon            004 033
input              001
integer            007
newton             001
number             007 014 016 017 019 023 024 029 033 035
output             001
read               014
real               008
root               008 027 029 029 029 030 031 033
sqr                033
sqroot             008 023 024 031 031
sqrt               023
write              013
writeln            012 017 020 024 025 030

                     0 instructions generated.
                  0.00 seconds total code generation time.
```

Assignment #2

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