CMPE 152: Compiler Design

September 19 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

Assignment #3

- Invent a new Pascal **when** statement:

```
WHEN
    i = 1 => f := 10;
    i = 2 => f := 20;
    i = 3 => f := 30;
    i = 4 => f := 40;
    OTHERWISE => f := -1
END
```

- New syntax, but old semantics, equivalent to:

```
IF      i = 1 THEN f := 10
ELSE IF i = 2 THEN f := 20
ELSE IF i = 3 THEN t := 30
ELSE IF i = 4 THEN f := 40
ELSE               f := -1;
```

Assignment #3, *cont'd*

- New syntax:
  - Any boolean expression as the selector to the left of **=>**
  - A single statement (which can be compound) to the right of **=>**
- Old semantics:
  - Evaluate the boolean selectors sequentially from first to last.
  - If a selector is true, then execute the corresponding statement to the right of **=>** and then leave the statement.

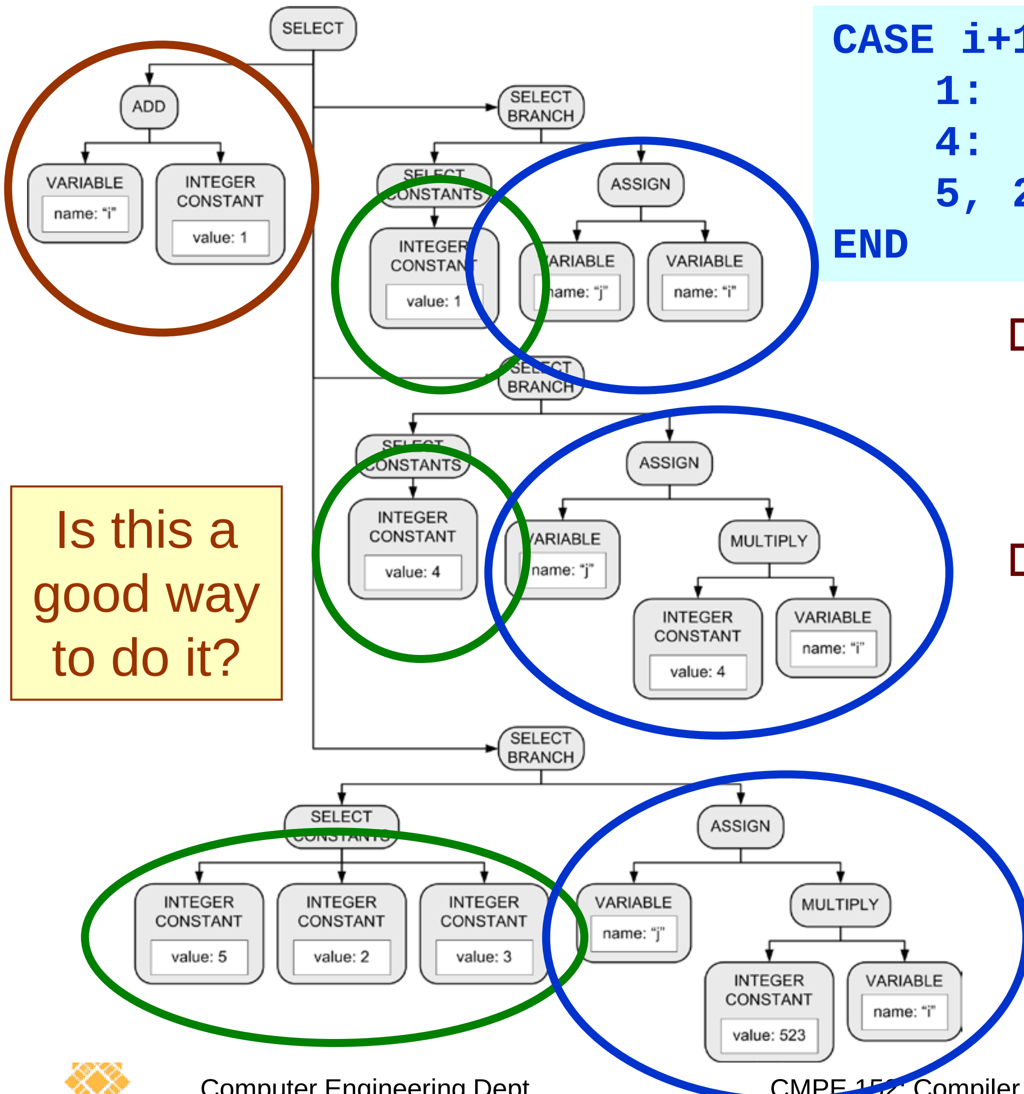
Assignment #3, *cont'd*

- Draw syntax diagrams for the **when** statement.
- Design the parse tree for the **when** statement.
  - Draw the tree for a simple **when** statement.
- Write the parser for the **when** statement.
  - Test it on some sample **when** statements.
  - Is it building proper trees?

Assignment #3, *cont'd*

- Write the backend executor for the **when** statement.
  - Test it by executing some sample statements.
  - Do you get the expected results?
- Due Monday, October 2.

Executing a SELECT Parse Tree



Is this a good way to do it?

```
CASE i+1 OF
    1:        j := i;
    4:        j := 4*i;
    5, 2, 3: j := 523*i;
END
```

- Evaluate the first child expression subtree to get the selection value.
- Examine each SELECT BRANCH subtree.
  - Look for the selection value in the SELECT CONSTANTS list of each SELECT BRANCH.
  - If there is a match, then execute the SELECT BRANCH's statement subtree.

Executing a SELECT Parse Tree, *cont'd*

- Why is searching for a matching selection value among all the SELECT BRANCHes bad?
  - It's inefficient.
- Selection values that appear earlier in the parse tree are found faster.
  - The Pascal programmer should not have to consider the order of the selection values.

Executing a SELECT Parse Tree, *cont'd*

- A better solution:
  For each SELECT tree, create a jump table implemented as a hash table.

- Build the table from the SELECT parse tree.

- Each jump table entry contains:
  - **Key:** A constant selection value.
  - **Value:** The root node of the corresponding statement subtree.

Executing a SELECT Parse Tree, *cont'd*

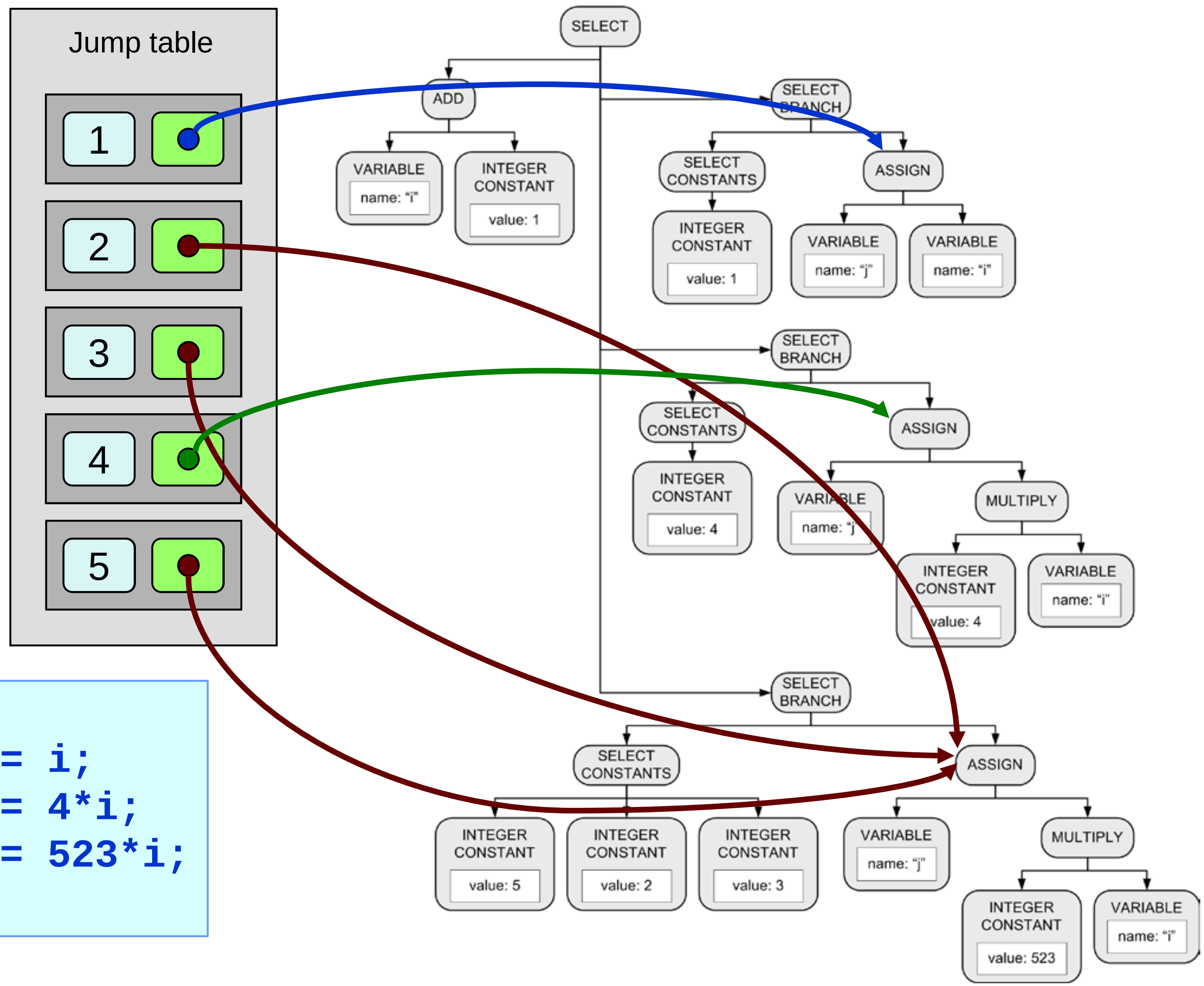
- During execution, the computed selection value is the key that extracts the corresponding statement subtree to execute.

SELECT Jump Table

Key: constant selection value

Value: root node of the corresponding statement

This is an example of **optimization** for **faster execution**.

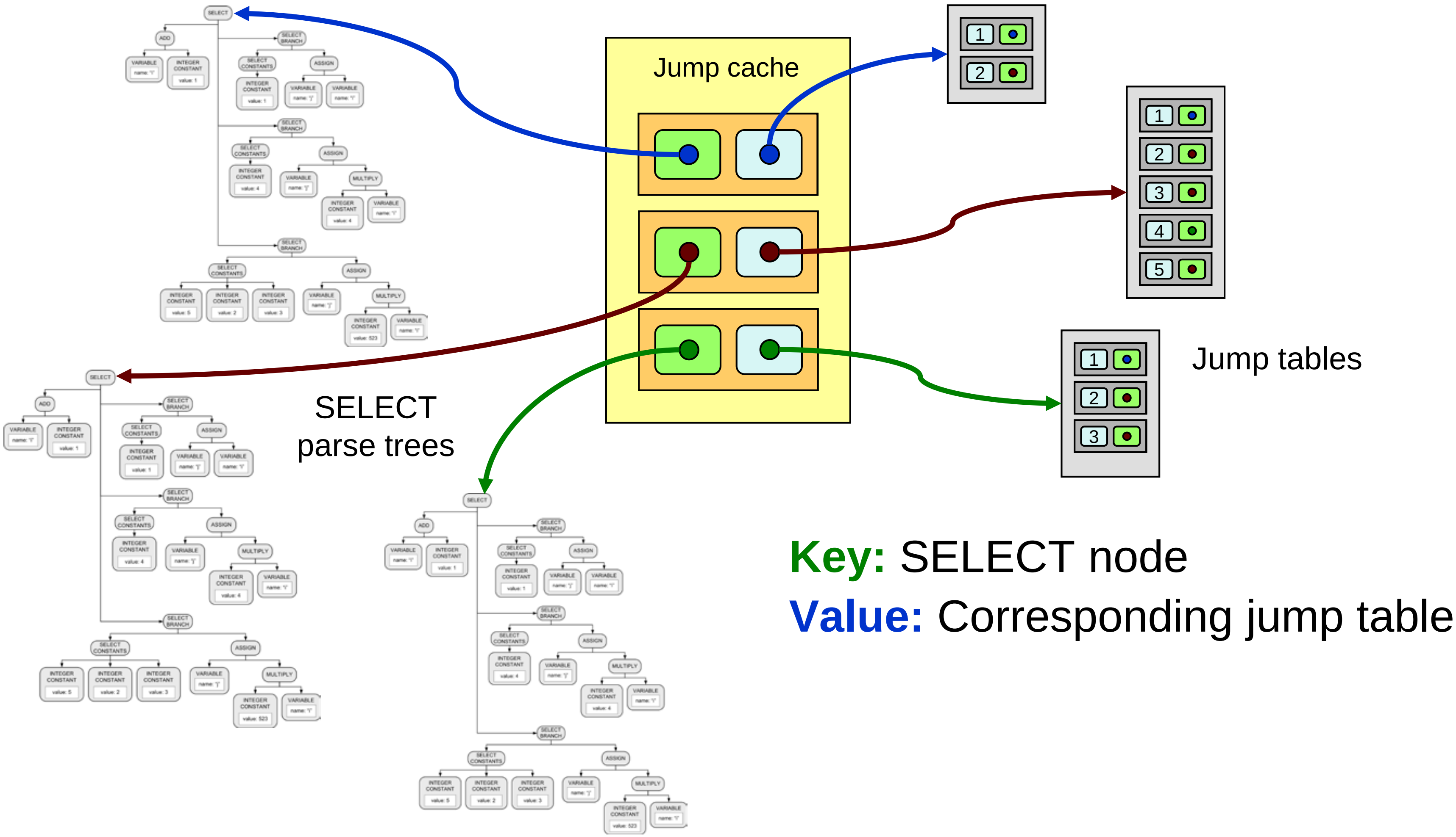



```
CASE i+1 OF
    1:        j := i;
    4:        j := 4*i;
    5, 2, 3: j := 523*i;
END
```

Multiple **CASE** Statements

- If a Pascal source program contains multiple **CASE** statements, there will be multiple SELECT parse trees.
- Create a global jump cache, a hash table of jump tables.
- Each jump cache entry contains:
  - **Key :** The SELECT node of a SELECT parse tree.
  - **Value:** The jump table created for that SELECT parse tree.

Jump Cache of Jump Tables



Key: SELECT node

Value: Corresponding jump table

Class SelectExecutor

- The global jump cache, which contains a jump table for each SELECT tree in the Pascal source program.

```
// Jump cache: entry key is a SELECT node,
//             entry value is the jump table.

// Jump table: entry key is a selection value,
//             entry value is the branch statement root node.

typedef map<int, ICodeNode *> JumpTable;
typedef map<ICodeNode *, JumpTable *> JumpCache;
```

Class SelectExecutor

```
DataValue *SelectExecutor::execute(ICodeNode *node)
{
    JumpTable *jump_table = jump_cache[node];
    if (jump_table == nullptr)
    {
        jump_table = create_jump_table(node);
        jump_cache[node] = jump_table;
    }

    vector<ICodeNode *> select_children = node->get_children();
    ICodeNode *expr_node = select_children[0];

    ExpressionExecutor expression_executor(this);
    DataValue *select_value = expression_executor.execute(expr_node);

    ICodeNode *statement_node = (*jump_table)[select_value->i];
    if (statement_node != nullptr)
    {
        StatementExecutor statement_executor(this);
        statement_executor.execute(statement_node);
    }

    ++execution_count;  // count the SELECT statement itself
    return nullptr;
}
```

Get the right jump table from the jump cache.

Evaluate the selection value.

Get the right statement to execute.

Can we eliminate the jump cache?

Simple Interpreter II

- Demos
  - `java –classpath classes Pascal execute case.txt`

Multipass Compilers

- A compiler or an interpreter makes a "pass" each time it processes the source program.
  - Either the original source text, or
  - The intermediate form (parse tree)

Three-Pass Compiler

- We've designed a 3-pass compiler or interpreter.
- **Pass 1:** Parse the source in order to build the parse tree and symbol tables.
- **Pass 2:** Work on the parse tree to do some optimizations.
  - Example: Create **CASE** statement jump tables.
- **Pass 3:** Walk the parse tree to generate object code (compiler) or to execute the program (interpreter).

Multipass Compilers, *cont'd*

- Having multiple passes breaks up the work of a compiler or an interpreter into distinct steps.
- Front end, intermediate tier, back end:
  - Modularize the structure of a compiler or interpreter
- Multiple passes:
  - Modularize the work of compiling or interpreting a program.

Multipass Compilers, *cont'd*

- Back when computers had very limited amounts of memory, multiple passes were necessary.
- The compiler code for each pass did its work and then it was removed from memory.
- The code for the next pass was loaded into memory to do its work based on the work of the previous pass.

Multipass Compilers, *cont'd*

- Example: The FORTRAN compiler for the IBM 1401 could work in only 8K of memory and made up to 63 passes over a source program.
  - See:

http://www.cs.sjsu.edu/~mak/CS153/lectures/IBM1401FORTRANCompiler.pdf

Scripting Engine

- We now have a simple scripting engine!
  - Expressions
  - Assignment statements
  - Control statements
  - Compound statements
  - Variables that are untyped

What's Next?

- Parse Pascal declarations
- Type checking
- Parse procedure and function declarations
- Runtime memory management
- Interpret <u>entire</u> Pascal programs.

Parsing Declarations

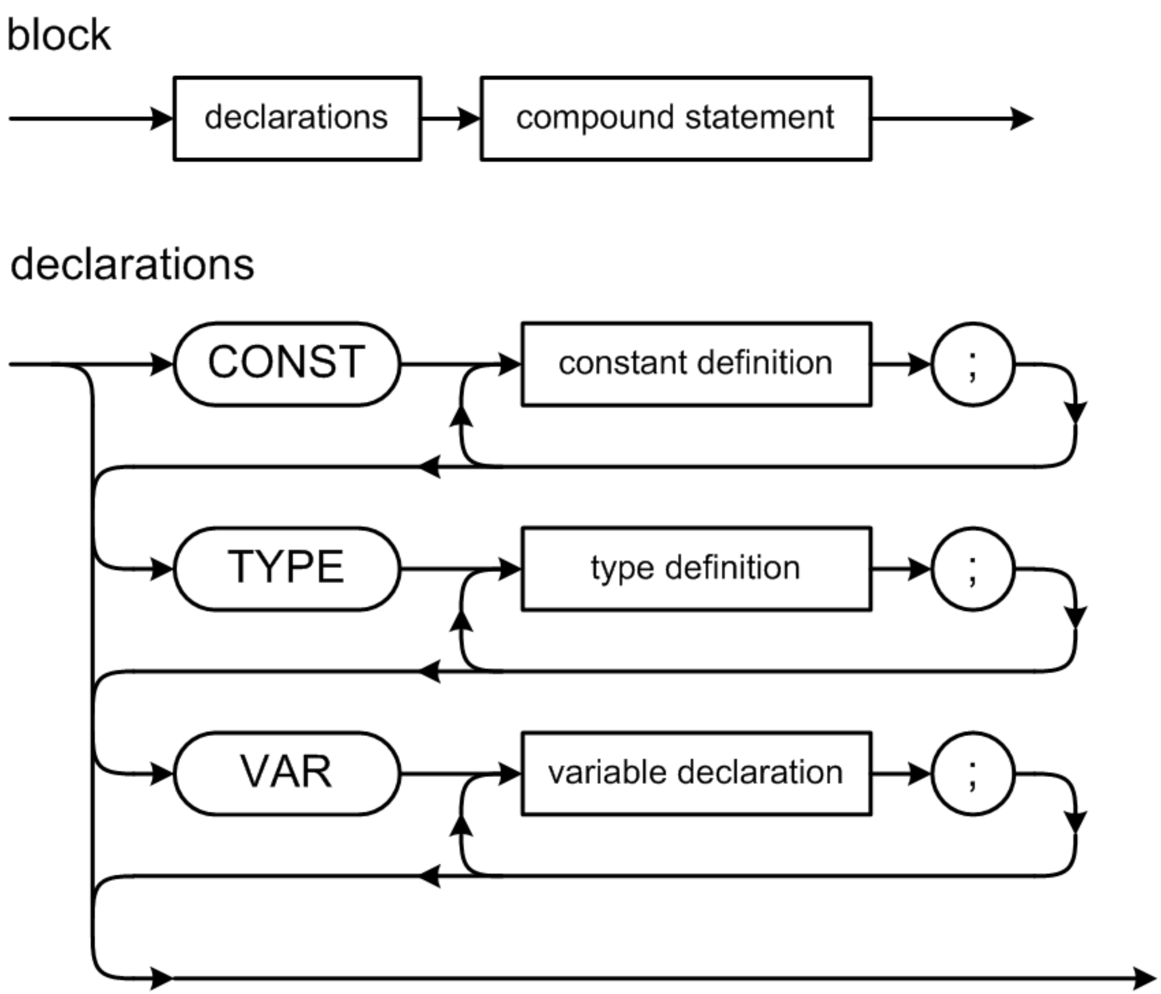
- The <u>declarations</u> of a programming language are often the <u>most challenging to parse</u>.

- Declarations syntax can be difficult.
- Declarations often include recursive definitions.
- You must keep of track of diverse information.
- Many new items to enter into the symbol table.

Pascal Declarations

- Classic Pascal declarations consist of 5 parts, each optional, but always in this order:
  1. Label declarations
  2. Constant definitions
  3. Type definitions
  4. Variable declarations
  5. Procedure and function declarations
- We will examine 2, 3, and 4 next.
  - We'll do procedures and functions in a couple of weeks.

Pascal Declarations



- The **CONST**, **TYPE**, and **VAR** parts are optional, but they must come in this order.
- Note that constants and types are **defined**, but variables are **declared**.
- Collectively, you refer to all of them as declarations.