

# CS331 Compiler Design

CS331 • Introduction

## Overview

- We will examine the application of the theoretical constructs covered in CS240
  - develop programs for translating computer programs written in a high-level language into a form suitable for execution
- Build front end of a compiler for a subset of the Pascal language

CS331 • Introduction

## Translators

- Translate a program written in a **source language** into **object language**
  - Both source and object are “artificial” languages



CS331 • Introduction

## Compiler as Translator

- Source language is a **high-level** programming language (e.g. C, C++, Java, Pascal, Fortran, etc.)
- Object language is a **low-level** language e.g., assembly language or machine language
- **Functional equivalence:** source and object algorithms must be identical
  - Same output for a given input

CS331 • Introduction

## Translation

- Artificial translation rapidly became a mathematical discipline
- Overall process:
  1. Grasp exact meaning of each source “sentence”
    - **Parsing** : uncovering meaning and structure of source
  2. Compose an equivalent sentence in the object language
    - Perform transformations on the structure to yield object program

CS331 • Introduction

## Object Possibilities

- **Assembly language**
  - Requires another translation by the **assembler** to machine language
  - Easy to generate
    - Simple structure
      - No nested statements, complex arithmetic expressions, higher level control, procedures
    - Fixed format
      - A few fixed fields (instruction field, address field)
    - One assembly language statement per machine instruction

CS331 • Introduction

- **Machine code**
  - Binary instructions
    - “re-locatable object code”
  - Advantage : can be executed directly
- **Execution**
  - Translate source program to intermediate data structure and execute the instructions
  - This kind of translator known as an **interpreter**
- ...and others
  - Java compiler translates from Java to interpretable bytecode

CS331 • Introduction

## Why Do We Need Translators?

- Enables use of **high-level languages**
- Otherwise, required to use machine languages
  - expressed in 1’s and 0’s
  - deal directly with hardware (e.g.registers)

CS331 • Introduction

## Evolution of Programming Languages

1. **Machine language**
2. **Symbolic assembly language**
  - “mnemonics”: names for memory locations instead of addresses
3. **Assembler macros**
  - One statement for many
4. **High-level languages**
  - Machine independent
  - Natural notation
  - Instruction explosion

CS331 • Introduction

## Source Code

- Optimized for **human readability**
  - **expressive**: matches human notions of grammar
  - **redundant** to help avoid programming errors

```
int expr(int n)
{
    int d;
    d = 4 * n * n * (n + 1) * (n + 1);
    return d;
}
```

CS331 • Introduction

## Machine code

- Optimized for **hardware**
  - Redundancy, ambiguity reduced
  - Information about intent lost
  - Assembly code  $\approx$  machine code

```
lda $30, -32($30)      ldl $3, 16($15)
stq $26, 0($30)       addq $3, 1, $4
stq $15, 8($30)       mull $2, $4, $2
bis $30, $30, $15     ldl $3, 16($15)
bis $16, $16, $1      addq $3, 1, $4
stl $1, 16($15)       mull $2, $4, $2
lds $f1, 16($15)      stl $2, 20($15)
stq $f1, 24($15)     ldl $0, 20($15)
ldl $5, 24($15)      br $31, $33
bis $5, $5, $2       $33:
s4addq $2, 0, $3      bis $15, $15, $30
ldl $4, 16($15)       ldq $26, 0($30)
mull $4, $3, $2       ldq $15, 8($30)
                    addq $30, 32, $30
                    ret $31, ($26), 1
```

CS331 • Introduction

## Low-Level Languages

- **Machine Language** (Binary)
  - ☹ Machine friendly / user hostile ☹
  - Tightly coupled to The Machine
  - Very terse
- **Assembly Language**
  - Mnemonic version of machine language
    - Access to all supported instructions and formats
  - Features
    - Registers
    - Labels
    - Mnemonics
    - Storage control
    - Potential for highly efficient use of hardware
  - Liabilities
    - Little program structure – highly error prone
    - No reusability to other instruction sets
    - Terribly expensive to program this way

CS331 • Introduction

## Higher-Level Languages

- **Goals of high level language**
  - **Notational convenience** with appropriate “expressibility”
  - **Machine independence** (reuse, portability)
  - **Human friendly**
  - **Easy maintenance**
  - **Machine translation** to target environment
    - Appropriate granularity of operators and objects
  - May support an **abstract programming environment**
    - distributed? concurrent? secure?
- Multiple **families of higher-level languages**
  - Imperative
  - Object-Oriented
  - Functional
  - Logical

CS331 • Introduction

## Imperative Languages

- Action Oriented
- Fortran
  - Formula Translation
  - Numerical/Scientific Computing
  - 1958
- Also called **procedural**, since one describes the computation by detailed procedures

CS331 • Introduction

## Evolution of Imperative Languages

- **Algol** (“The Algol-60 Report”)
  - 1960
- **PL/1** (interpreter and compiler)
- **Pascal**
  - Teaching Language
- **C** (AT&T)
  - Systems Programming
  - Popular after Unix was rewritten in C
- Imperative languages extend to greater structure as object-oriented languages

CS331 • Introduction

## Object Oriented

- Encapsulate data and procedures together
- Extend abstract data types by inheritance to allow type/subtype relationships
- Inheritance hierarchy defines type/subtype relationship
- Virtual functions (in C++) define type dependent operations within the hierarchy

CS331 • Introduction

## Logical-based languages

- **Prolog**
  - Programming in Logic, 1972
  - Domains include natural language processing
- Resolution theorem prover makes “all” valid inferences (not procedural)
  - Programmer does not write “control structure”
  - Express as logical prepositions and facts
  - Impure “cut” operators let programmer direct the inference process

CS331 • Introduction

## Functional languages

- Specify functions
  - Decompose into smaller functions
  - (Often) a single data type
  - Should not have side effects
- Self referential, functions are first class objects
  - program can easily create new expressions and execute its data

CS331 • Introduction

## Functional Languages

- **Lisp** (the cool language!)
  - List Processing
  - 1958
  - See McCarthy report in Library
  - Car/Cdr/Cons/Cond,  $\lambda$ -calculus
- **Scheme**
  - A brand of LISP

CS331 • Introduction

## Language Definition

- **Fortran** described by an informal document (several hundred pages)
- **Algol** described by formal (context-free) grammar with English semantics (15 pages)

**The first Fortran compiler took 18 man-years to build!**

CS331 • Introduction

## Two paradigms for language processors

- **Interpreter**
  - Efficient for prototyping (rapid prototyping)
  - Efficient error reporting
  - Dynamic debugging
- **Compiler**
  - Efficient for production applications
  - Order of magnitude faster

CS331 • Introduction

## Interpreter

- Target is high-level machine or program
  - Typically a **virtual machine**
    - Provide extended runtime capabilities
    - May also provide flexible execution environment
  - Processes source-code or intermediate-code
    - Reinterpret each statement every time
    - Eliminates the “syntactic sugar” of specific syntax
    - Supports symbol table and storage management
    - May support optimization through dynamic program properties
- Examples
  - Lisp runs with simple interpreter
  - Java runs in portable Java Machine (JVM)

CS331 • Introduction

## Compiler

- Target is lower-level machine, typically **assembler**
  - One-time transformation and optimization for underlying hardware (or other runtime model)
  - Machine-independent internal forms
  - Machine-dependent output
- Syntax-directed verification (well-formed programs)
- Translation and optimization for underlying hardware
  - Semantic enforcement
  - Optimization
- Leverage hardware knowledge for efficient runtime
  - scheduling, pipelines, caches, etc.

CS331 • Introduction

## Hybrid Processors

- **Hybrid** (Compiled-Interpreted)
  - Java
    - Convert to Bytecode (portable code)
    - Interpret Bytecode
    - Just In Time (JIT) compiler
      - code generator that converts Java bytecode into machine language instructions
      - code runs much faster than interpreted code
      - some Java VMs include both an interpreter and JIT

CS331 • Introduction

## How to translate?

- Source code and machine code mismatch
- Some languages farther from machine code than others (“higher-level”)
- **Goal:**
  - source-level expressiveness for task
  - best performance for concrete computation
  - reasonable translation efficiency
  - maintainable code

CS331 • Introduction

## Correctness

- Programming languages describe computation precisely
- Therefore: translation can be precisely described
- **Correctness** is very important!
  - hard to debug programs with broken compiler...
  - non-trivial: programming languages are expressive
  - implications for development cost, security

**This course:  
Techniques for building correct compilers**

CS331 • Introduction

## Language Design Issues for Compilation

- **Form of names, statements**
  - Blanks allowed? Fortran : DO I 10 = ...
- **Scope of names**
  - **Block-structure** vs. **non-block structure**
  - Reference to a name requires consulting table for names known (declared) in that block
  - Names not available must be kept separate
  - “most closely nested” rule

CS331 • Introduction

- **Dynamic vs. static allocation**
  - Is storage mapped out at compilation time, or determined at run-time? (different code)
- **Binding of identifiers to names**
  - **Identifier** : user-specified string
  - **Name** : compiler-designated object with specific attributes
    - Name is bound to storage location
- **Binding to type : three possibilities**
  - All variables declared and type specified
  - Type determined from form of name
  - Type determined from context

CS331 • Introduction

- **Parameter passing**

- Value
- Reference
- Value-result
- Name
- Constant

- **Recursion**

- allocate storage for each local instance of variables

CS331 • Introduction

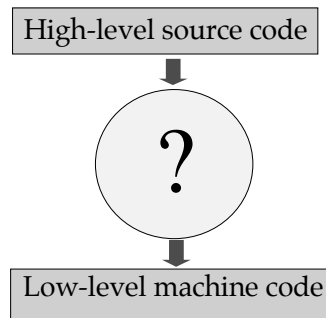
(Aside)

## The Pass by Name Problem

```
procedure swap(x,y);  
integer x, y;  
begin  
  integer t;  
  t := x;  
  x := y;  
  y := t;  
end;  
  
Call swap(A[i],j):  
begin  
  integer t;  
  t := A[i];  
  A[i] := j;  
  i := t;  
end;  
  
Call swap(i,j):  
begin  
  integer t;  
  t := i;  
  i := j;  
  j := t;  
end;  
  
Call swap(j, A[i]):  
begin  
  integer t;  
  t := i;  
  i := A[i];  
  A[i] := t;  
end;
```

CS331 • Introduction

## How to translate effectively?

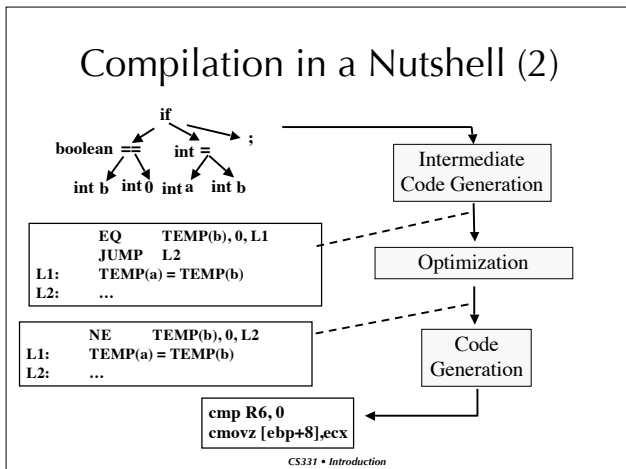
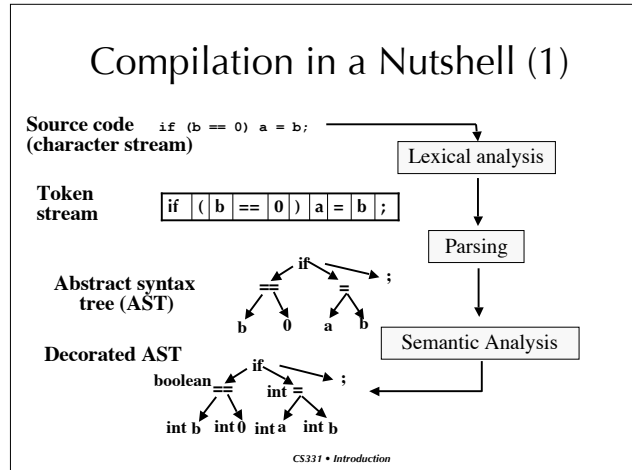
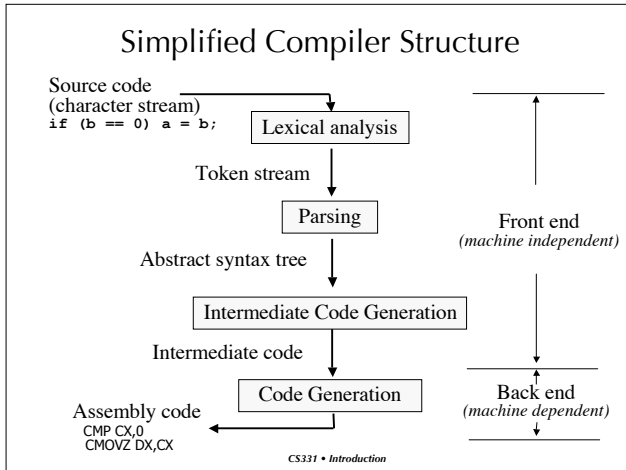


CS331 • Introduction

## Idea: Translate in Steps

- Series of program representations
- Intermediate representations optimized for program manipulations of various kinds (checking, optimization)
- More machine-specific, less language-specific as translation proceeds

CS331 • Introduction



- ### Other Compiler Pieces
- **Symbol table manager**
    - "bookkeeper"
    - Maintains names used in program and information about them
      - Type
      - Kind : variable, array, constant, literal, procedure, function, record...
      - Dimensions (arrays)
      - Number of parameters and type (functions, procedures)
      - Return type (functions)
      - Etc.
- CS331 • Introduction

- **Error handler**

- Control passed here on error
- Provides information about type and location of error
- Called from any of the modules of the front end of the compiler
  - **Lexical errors** e.g. illegal character
  - **Syntax errors**
  - **Semantic errors** e.g. illegal type

CS331 • Introduction