Welcome to CSC488/2107
Language Processors

Instructor	Prof Marsha Chechik
	Pratt Building, Room 384, chechik@cdf
Lectures	Tuesday 10-11 Lash Miller 161
	Thursday 10-11 Lash Miller 161
Tutorial	Thursday 9-10 Lash Miller 161
TAs	Iluju Kiringa
	Xuan Chen
	Hao (Richard) Zhang
Office Hours	Tuesday 1-2, Thursday 11-12 and by appointment

Admin (Cont.)

Text	Charles Fischer and Richard J. LeBlanc Jr.
	Crafting a Compiler in C, Benjamin/Cummings 1991
Marking	Midterm test (15%), Final Exam(30%), Course Project(55%)
News Group	`ut.cdf.csc488h` **You are required to read it!**
Web Page	`http://www.cs/~chechik/courses97/csc488`
Slides	`http://www.cs/~chechik/courses97/csc488/notes.html`

Course Project

- Design and Implementation of a small compiler system.
- Work in teams of 4 (- 1 , + 0)
- Six Phase Project

Phase 1	Write programs in project language
Phase 2	Revise translation grammar to LR(1)ness (using Yacc)
Phase 3	Implement symbol table and partial semantic checking.
Phase 4	Design code generation, complete semantic checking.
Phase 5	Implement code generator
Phase 6	Add more language features

- Project specification announced soon

Immediate Project Issues

- Team Selection
  Work in teams of 4, self-selection of teams
  Form teams *ASAP*. List due on January 13
  Team management and conflict resolution is YOUR responsibility
  Instructor/tutor deal with orphans
- Project will be done in C
- **Selecting a hard-working, compatible team is important for success in the course project**
- Everyone on team gets the same mark for project

Other Project Issues

- Your assignments should be well documented and well-tested (see handout).
- We emphasize use of tools: `CVS`, `lex` and `yacc`.
  Some overview done in tutorial, details - on your own
- Course is fast-paced, and we get started on the project almost right away

Graduate Students

- Need not do extra work for the project
- Will have extra/replacement problems in midterm and exam

Course Outline

Compiler structure	Chapter 1
Lexical Analysis	Chapter 3
Syntax Analysis	Chapters 4, 5, 6
Tables & Dictionaries	Chapter 8
Semantic Analysis	Chapters 7, 10, 11
Translation techniques	Chapters 12, 13
Code generation	Chapter 9, 15
Optimization	Chapters 15, 16
Special topics	

Course Schedule

Jan 8	Description of assignments available
Jan 13	List of teams due
Jan 20	Assignment 1 due
Jan 29	Assignment 2 due
Feb 3	Solutions to Assignment 2 available
Feb 24	Assignment 3 due
Feb 26	Midterm
Mar 3	Assignment 4 due
Mar 6	Drop deadline
Mar 17	Assignment 5 due
Apr 7	Assignment 6 due

Reading Assignment

`Fischer and LeBlanc,`

`Chapter 1`

What Do Compilers Do?

- Check source program for correctness
  - Well formed lexically
  - Well formed syntactically.
  - Satisfies static semantic checks

    Type correctness

    Usage correctness

    Implementation limits
- Transform *source program* into an executable *object program*

What Should a Compiler Implementor Know?

- A large *variety* of programming languages (CSC32[46]H)
- File and Data structures (CSC228H,CSC378H or ECE242H)
- Computer organization (CSC 258H or ECE341H/ECE353H)
- Software engineering (CSC 408H or CSC442H)
- Software Tools (CSC209H)
- Compiler implementation techniques (CSC488H, ECE540H).
- Communication Skills (ENG 182H)
- Some operating systems (CSC468H or ECE344H/ECE443H)

Compiler Writing Requires Analytic Skills

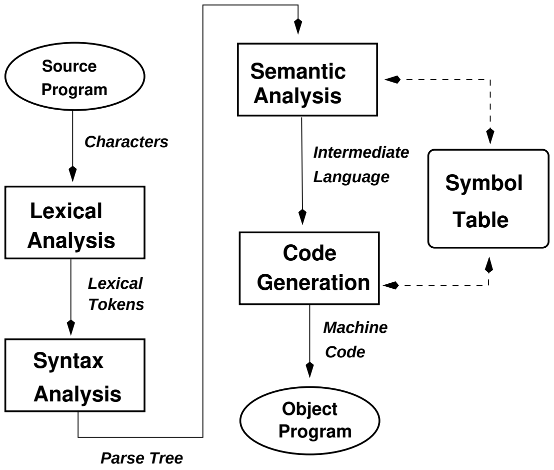
- The compiler writer designs the mapping from the source language to the target machine.
- Must be able to analyze a programming language for potential problems. Determine if language can be processed during lexical analysis, syntax analysis, semantic analysis and code generation.
- Must be able to analyze target machine and determine best way to implement each construct in the programming language.
- A production compiler generally must implement the *entire* language. Student project and prototype compilers often omit the hard parts.

Programming Language Designers are the Enemy

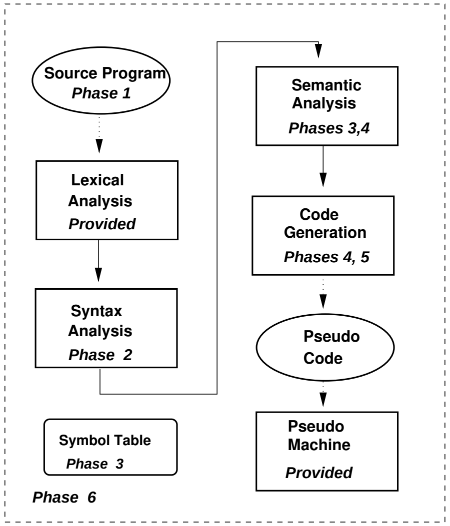
- Most programming language definitions are incomplete, imprecise and sometimes inconsistent.
- Language designers usually don't think much about the details of the implementation of a language, leaving lots of problems for the compiler writer.
- Examples of language implementation problems.
  - Fortran syntax DO 10 I = 1 , 10
  - DO loop in PL/I DO v = ei TO el BY es
  - Iterative loops in most languages: **for** i := ei **to** maxInt **do**
  - Pascal and Turing integer subrange notation: 1 **..** 9
  - String operations in most languages:

    SUBSTR(stringExpn , startExpn , endExpn)
  - Aliasing detection in old Turing.

Generic Compiler



Project Preview



Compiler Design Issues

- Like any large, long-lived program a compiler should be designed in a modular fashion that is easy to maintain over time.
- Need to design a software architecture for the compiler that allows it to implement the required language processing steps.
- Architecture of the compiler will be influenced by
  - The programming language being compiled.
  - Characteristics of the target machine(s).
  - Compiler design goals
  - Compiler's operating environment.
  - Compiler project management goals

Characteristics of a Good Compiler

- User Interface
  - Precise and clear diagnostic messages
  - Easy to use processing options.
- Correctly implements the language
- Detects all *statically* detectable errors.
- Generates highly optimal code.
- Compiles quickly using modest system resources.
- Compiler Structure
  - Well modularized. Low coupling between modules.
  - Well documented and maintainable.
  - High level of internal consistency checking.

Programming Language Influences

- Declaration before use?
- Typed or typeless?
- Separate compilation? modules/objects?
- Lexical issues, designed to be lexable?
- Syntactic issues, designed to be parseable?
- Static semantic checks required? Implementable?
- Run-time checking required?
- Size of programs to be compiled?
- Compatibility with OS or other languages?
- Dynamic creation/modification of programs?

Target Machine Influences

- Limited or partitioned memory
- RISC vs. CISC instruction set.
- Irregular or incomplete instruction set.
- Inadequate addressing modes.
- Hardware support for run-time checking?
- Poor support for High Level languages.
- Missing instruction modes?
- Inadequate support for memory management?

Some Compiler Design Goals

- **Correctly implement the language.**
- Be highly diagnostic and error correcting.
- Produce time or space optimized code.
- Be able to process very large programs.
- Be very fast or very small.
- Be easily portable between environments.
- Have a user interface suitable for inexperienced users.
- Emit high quality code.

Some Project and Environment Goals

- Project Goals
  - Minimize compiler development time.
  - Minimize compiler development budget.
  - Minimize compiler project staffing.
  - Minimize compiler project resource usage.
- Environment Goals
  - Run on machine M using operating environment S.
  - Run on machine X and generate code for machine Y.
  - Run in no more than Z bytes of memory.

Example Compiler Goals

- Student Compiler
  - Interface for inexperienced users.
  - Be highly diagnostic at compile time and run-time.
  - Compile with blinding speed.
  - Do *no* optimization
- Production Compiler
  - Interface for experienced users.
  - Produce highly optimized object code.
- Quick and Dirty Compiler
  - Minimize compiler construction time
  - Minimize project resource usage and budget.
  - Do no optimization, omit hard parts of language.
  - Compile to interpretive code, assembly language or high-level language.

Some Compiler Design Choices

- Organization of compiler processing
  - Single pass or multiple pass?
- Choice of compiler algorithms
  - Lexical and syntactic analysis
  - Static semantic analysis, code generation
  - Optimization
- Compiler data representation
  - Symbol and/or type tables, dictionaries
  - Memory resident compiler data?
  - Communication between passes?
  - Format of compiler output

Getting the compiler structure right the first time is essential to project success.

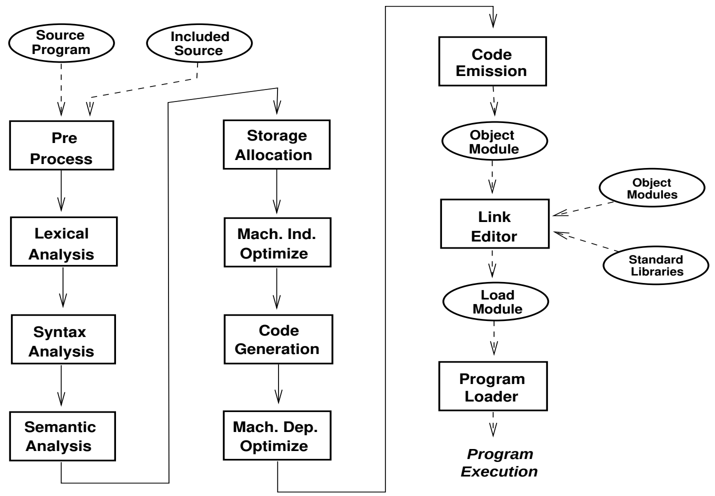
Some Compiler Design Choices

- Form of compiler output
  - Absolute machine code
  - Relocatable machine code
  - Object module
  - Load module
  - Assembly language
  - Interpretive pseudo code
  - High level programming language

Interpretive Systems

- Pseudo machine code that is a simple encoding of the program.
- The pseudo machine code is executed by another program (an *interpreter*)
- Advantages of interpretors
  - Debugging newly written programs.
  - Student compilers that require good run-time error messages.
  - Languages that allow dynamic program modification.
  - Typeless languages that can't be statically analyzed for proper semantics.
  - Cases where run-time size must be minimized.
  - Implementing ugly language features.
  - Quick and dirty compilers.
  - As a way to port programs between environments.
- Disadvantages of interpretors
  - Execution speed, usually at least 100 times slower than machine code.
  - May limit user data space or size of programs.
  - May require recompilations for each run.

Canonical Compiler Processing



Single Pass Compilers

- Good for simpler languages (e.g. Pascal) and "Quick and Dirty" implementations (e.g. prototypes and course projects).
- Not feasible for languages that permit backward information flow (e.g. declaration *after* use).
- Single pass compilers are usually parser-driven.
  The parser coroutines with the lexical analyzer to obtain tokens.
  The parser calls semantic analysis and code generation routines as each construct in the language is parsed.
- By the time a program is completely parsed, all semantic checks have been performed and all code has been generated.
- Some parsers can only invoke semantic and code generation actions after an entire construct (e.g. statement) has been recognized. Other parsers allow more flexibility in the placement of these actions.

Single Pass Example

statement	$\rightarrow$	ifStatement
ifStatement	$\rightarrow$	**if** expression *Sbool Tbrf* truePart falsePart **fi**
truePart	$\rightarrow$	**then** statement
falsePart	$\rightarrow$	λ *Tfixbf*
	|	**else** *Tbr Tfixbf* statement *Tfixbr*

Actions

Sbool	Semantic check that expression type is boolean
Tbrf	Emit a BranchFalse instruction. Save the address of this instruction
Tfixbrf	Fix up the address in a previously emitted branch false instruction to point at the next code address.
Tbr	Emit an unconditional branch instruction. Save instruction address.
Tfixbr	Fix up the address of a previously emitted branch instruction to point at the next code address

Multi Pass Compilers

- Multi pass compilers make several complete passes over the source program.
- There are many reasons why a compiler might be multi pass
  - Environment limits on compiler size.
  - Difficult or large language.
  - Compiler implementation constraints.
  - Size of source programs to be processed.
  - Degree of optimization.
  - Compiler project constraints.
  - Ease of implementation.

Multi Pass Compiler Issues

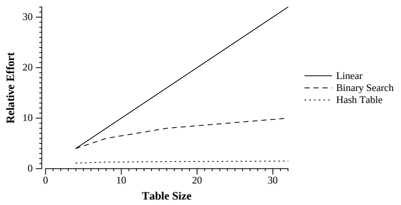
- Compiler implementation effort increases with the number of passes.
- Compiler speed decreases with the number of passes.
- A multi pass compiler does much more input and output to communicate between passes.
- Efficient internal representations of programs become more important as size of compiler/language increases.
  Memory resident tables are faster than disk resident tables.
- Partitioning a compiler into an efficient multi-pass structure is often a difficult software engineering task.
- Efficient compiler algorithms are important.

Interpass Communication Issues

- Information flows between compiler passes
  - Representation(s) of the program
  - Tables
  - Error messages
  - Compiler flags
  - Source program coordinates.
- Form of communication may change as program is processed.
  A compiler may use multiple representations of a program.
- Use disk resident information for large programs.
  Use memory resident information for compiler speed.
- **Backward information flow should be avoided if possible.**

Table & Dictionary Issues

● The symbol table and related tables are the central data structures used in the middle part of compiler processing.
● Efficient access to these tables is *crucial* to good compiler performance (speed).
● Table operations: create, modify, use, delete.
● Tables can be memory-resident or disk-resident.
● *Efficient table search is the dominant performance issue.*



Intermediate Languages

- Represent the structure of the program being compiled
  - Declaration structure.
  - Scope structure.
  - Control flow structure.
  - Source code structure.
- Used to pass information between compiler passes
  - Compact representation desirable.
  - Should be efficiently to read and write.
  - Provide utility to print intermediate language for compiler debugging.

Intermediate Language Examples

- Condensed Source

  A * B + C / D - E

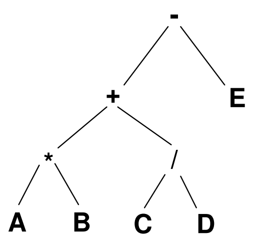
- Polish Postfix Notation

  A B * C D / + E

- Polish Prefix Notation

  - + * A B / C D E

- Parse Tree



More Intermediate Languages

- Quadruples

  (* , A , B , T1)
  (/ , C , D , T2)
  (+ , T1 , T2 , T3)
  (- , T3 , E , T4)

- Triples

  1 (* , A , B)
  2 (/ , C , D)
  3 (+ , (1) , (2))
  4 (- , (3) , E)

- Directed Acyclic Graphs

  Used to represent control structure

Compiler Architecture Examples

- Student compiler
  - One pass for speed
  - In-memory tables
  - Compile to directly executable absolute code or to interpretable code.
  - Tune for compile speed and high quality diagnostics.
- Production Optimizing Compiler
  - Usually multi-pass.
  - Uses disk-resident tables for large programs.
  - Usually includes heavy-weight optimization.
- Large Language, Small Computer
  - Many passes or phases.
  - All data stored on disk
  - Possibly emit interpretive code.

Table Driven Compilation

- Many compiler algorithms can be described in terms of a small set of actions that are executed in predictable orders.
- Table driven compilation techniques use this regularity to achieve small compiler size, ease of compiler modification and maintenance.
- Often there are special tools for generating the tables.
- Examples:

  lexical analysis (lex, flex),
  syntax analysis (yacc),
  code generation,
  semantic analysis
  CSC488S course project

Table Driven Compilation Strategy

- Determine distinct actions that must be performed.
- Determine execution sequence of actions.
  - May require conditional execution and iteration
  - Often describe executions sequences using a state machine.
- Table driven strategy:
  1. Assign an identifying number to each action.
  2. Encode execution sequences as a table of action identification numbers.
  3. Write code to perform each action.
  4. Write interpreter that reads input and executes actions based on entries in the table.

Table Driven Example

- Problem: Read C program and discard /* .. */ comments.
- Actions

A Copy current character to output, advance input

B Discard current character, advance input

C Emit character '\' to output.

- Action Table

State	Current Character	Next State	Action
1	/	2	B
	λ	1	A
2	*	3	B
	λ	1	C
3	*	4	B
	λ	3	B
4	/	1	B
	λ	3	B

λ is any other character.

State machine

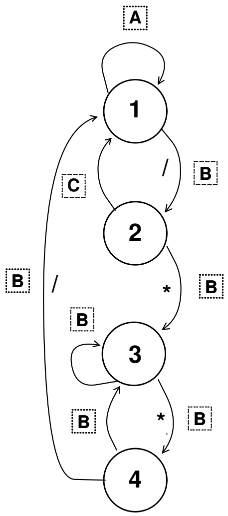


Table Interpreter Example

```
#define  A    1
#define  B    2
#define  C    3

struct stateStruct {
            char  matchChar   ;     /* match character         */
            char  matchState  ;     /* next state if match     */
            char  matchAction ;     /* next action if match    */
            char  elseState   ;     /* next state if no match  */
            char  elseAction  ;     /* next action if no match */
      }  stateTable[ 5 ]
       /*  StateTable initialized from table in previous slide */
         =   {   0,      0,     0,    0,    0   ,    /* 0 unused */
                 '/',    2,     B,    1,    A   ,    /* State 1 */
                 '*',    3,     B,    1,    C   ,    /* State 2 */
                 '*',    4,     B,    3,    B   ,    /* State 3 */
                 '/',    1,     B,    3,    B   };   /* State 4 */

struct stateStruct  * sp ;          /* pointer to state table  */
int   state, nextState, action ;    /* state variables         */
char  ch ;                          /* current input character */
```


Table Interpreter Example - cont'd.

```
ch = getchar() ;
state = 1 ;

while( TRUE ) {
    sp = & stateTable[ state ] ;
    if( ch == sp->matchChar ) {
        nextState = sp->matchState ;
        action    = sp->matchAction ;
    } else {
        nextState = sp->elseState ;
        action    = sp->elseAction ;
    }
    switch( action ) {
    case A :     putchar( ch ) ;
                 /*  FALL THROUGH */
    case B :     ch = getchar() ;
                 break ;
    case C :     putchar( ch ) ;
                 break ;
    } ;
    state = nextState ;
}
```