CSC 488S/CSC 2107S Lecture Notes

These lecture notes are provided for the personal use of students taking CSC488H1S or CSC2107S in the Winter 2012/2013 term at the University of Toronto

Copying for purposes other than this use and all forms of distribution are expressly prohibited.

CSC488S/CSC2107S - Compilers and Interpreters

Instructor	Prof. Dave Wortman		
	Bahen Centre, Room 5222 ,	dw@cdf.toronto.edu	
Lectures	Tuesday	14:00	WB 130
	Thursday	14:00	WB 130
Tutorial	Thursday	13:00	RS 208
	immediately after lecture and by appointment		
Text	Charles Fischer, Ron Cytron and Richard LeBlanc Jr. ,		
	Crafting a Compiler , Addison-Wesley 2009		
Marking	Mid term test, Final Exam, Course Project		
Web Page	`http://cdf.toronto.edu/~csc488h/winter/`		
Bulletin Board	**Read Often!!**		
	`https://csc.cdf.toronto.edu/csc488h1s`		
Slides	on the Bulletin Board		
Handouts	on the Bulletin Board		

Course Project

- Design and Implementation of a small compiler system.
- Work in teams of 5 (- 1 , + 0)
- Five Phase Project

Phase 1	Write programs in project language
Phase 2	Revise grammar and build parser
Phase 3	Implement symbol table and semantic checking.
Phase 4	Design code generation
Phase 5	Implement code generator

- Language specification and project details announced soon

Course Outline

Topic	Chapters
Compiler structure	Ch. 1, 2
Lexical Analysis	Ch. 3
Syntax Analysis	Ch. 4, 5, 6
Tables & Dictionaries	Ch. 8
Semantic Analysis	Ch. 7, 9
Run-time Environments	Ch. 12
Code generation	Ch. 11, 13
Optimization	Ch. 14

Reading Assignment

Fischer, Cytron, LeBlanc

Chapter 1

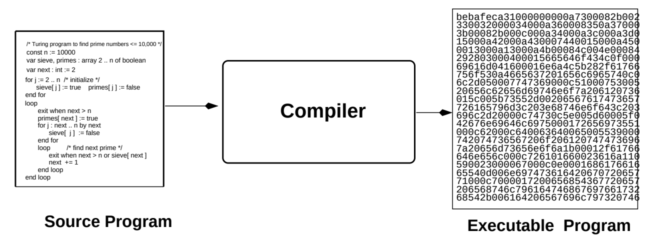
Section 10.1

What Do Compilers Do?

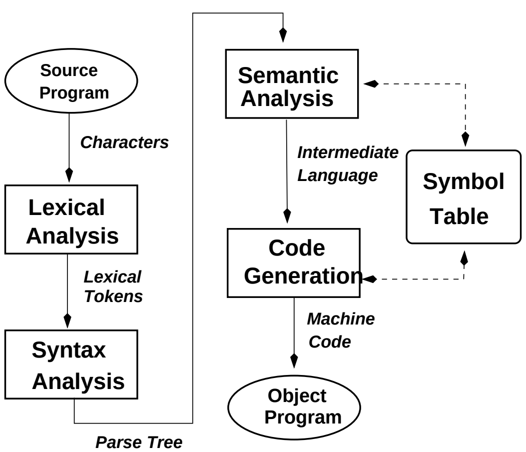
Check source program for correctness

- Well formed lexically
- Well formed syntactically.
- Passes static semantic checks
  - Type correctness
  - Usage correctness

Transform *source program* into an executable *object program*



Simple Generic Compiler



Lexical Analysis, Syntax Analysis

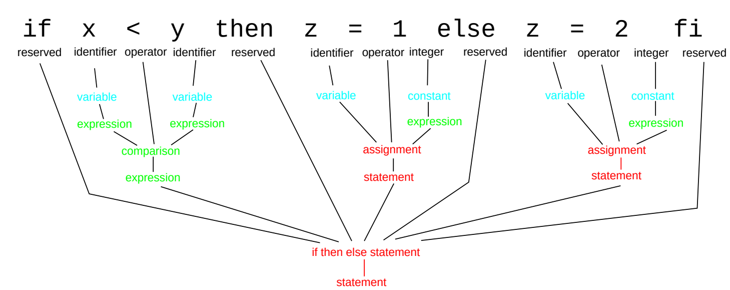
Source statement

```
if  x  < y  then  z  =  1  else  z  =  2   fi
```

Lexical analysis

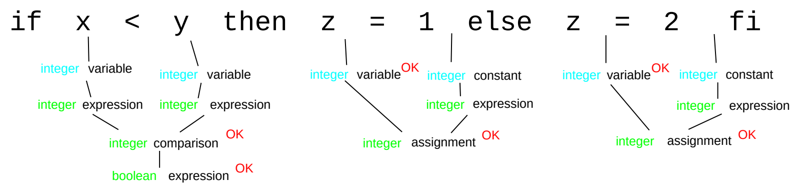
if	x	<	y	then	z	=	1	else	z	=	2	fi
reserved	identifier	operator	identifier	reserved	identifier	operator	integer	reserved	identifier	operator	integer	reserved

Syntax analysis



Semantic Analysis, Code Generation

Semantic analysis



Code Generation

```
if  x  <  y  then  z  =  1  else  z  =  2  fi

    load     r1,x        load      r1,=1      L23:  load      r1,=2
    load     r2,y        loadaddr  r2,z             loadaddr  r2,z
    less     r1,r2       store     r2,r1            store     r2,r1
    brfalse  L23         branch    L24        L24:
```


What Should a Compiler Implementor Know?

- Computer organization (CSC 258H)
- Software engineering (CSC 207H, CSC 301H, CSC 302H, CSC 410H)
- Software Tools (CSC 209H)
- File and Data structures (CSC 263H/CSC 265H)
- Communication Skills (CSC 290H)
- A large *variety* of programming languages (CSC 324H)
- Some operating systems (CSC 369H)
- Compiler implementation techniques (CSC 488H, ECE 489H).

Compiler Writing Requires Analytic Skills

- The compiler implementor(s) design 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.

Programming Language Designers are (usually) the Enemy

- Most programming language definitions are incomplete, imprecise and sometimes inconsistent. Real programs are written in language dialects.[a]
- Language designers often don't think deeply about the details of the implementation of a language, leaving lots of problems for the compiler writer.
- Typical problems
  - Poor lexical structure. May require extensive buffering or lookahead during lexical analysis
  - Difficulty syntax. Ambiguous, not suitable for normal parsing methods. May require hand written parser, backtracking or lookahead.
  - Incompletely defined or inconsistent semantics. User friendly options that are hard to implement.
  - Constructs that are difficult to generate good code for, make optimization difficult, require large run time support

[a] For a discussion of the difficulties of scanning and parsing real programs see `http://cacm.acm.org/magazines/2010/2/69354-a-few-billion-lines-of-code-later/fulltext`

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.
- A production compiler generally must implement the *entire* language. Student project and prototype compilers often omit the hard parts.
- 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

Programming Language Influences on Compiler Structure

- Declaration before use?
- Typed or type less?
- 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 on Compiler Structure

- 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?

Characteristics of an Ideal Compiler

- User Interface
  - Precise and clear diagnostic messages
  - Easy to use processing options.
- Correctly implements the entire 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.

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.

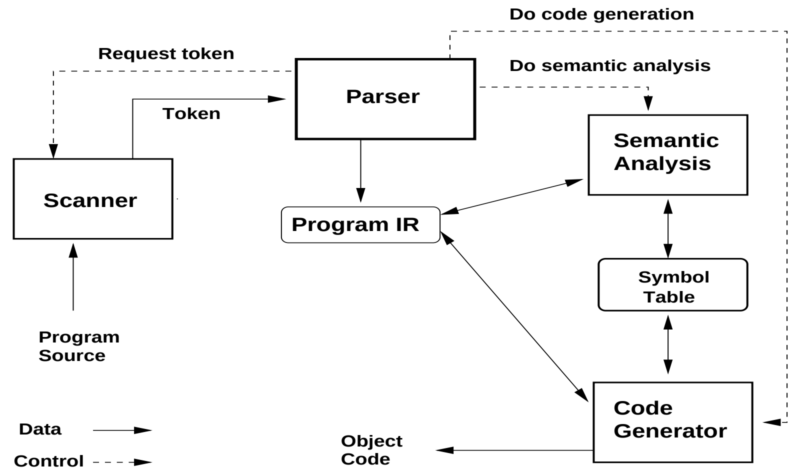
Example Compiler Goals

- 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.

Single Pass Compilers

- Good for simpler languages (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 each declaration or statement is completely parsed, all semantic checks have been performed and all code has been generated.
- Pascal is an example of a language well suited to single pass compilation.
  Declaration before use, well designed declaration structure, simple control structures.

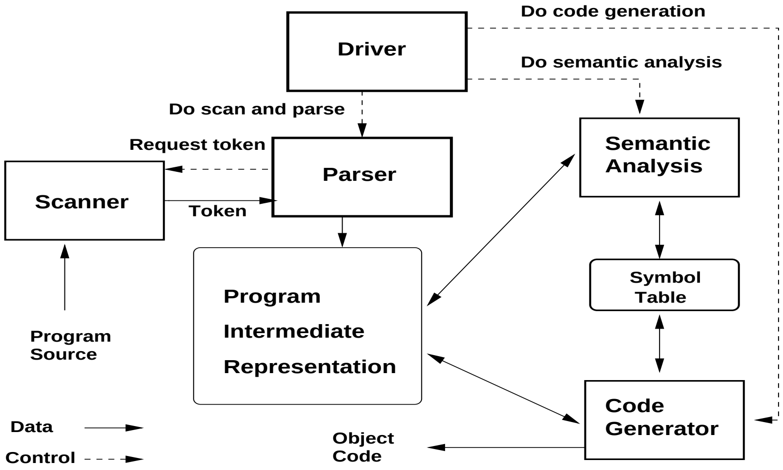
Single Pass Compiler Architecture



Multi Pass Compilers

- Multi pass compilers make several complete passes over the source program.
- Typically the first pass does lexical and syntax analysis and builds some intermediate representation of the program.
- Semantic analysis is a separate pass that processes this intermediate representation.
- Code generation is a separate pass that processes the intermediate representation.
- Optimization may make multiple passes over the program.
- Used for more complicated languages, e.g. Cobol, Ada, C, C++, Java
- Can couple lexical/syntax analysis for multiple languages to a common backend.

Multi Pass Compiler Architecture



Interpass Communication

- 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.**

Intermediate Representations

- 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 efficient 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 -

Triples

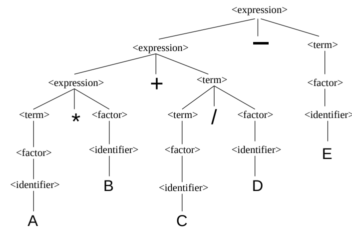
1 (* , A , B)
2 (/ , C , D)
3 (+ , (1) , (2))
4 (- , (3) , E)

Quadruples

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

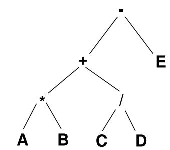
Parse Tree

Complete representation of the syntactic structure of the program according to some grammar.



Abstract Syntax Tree

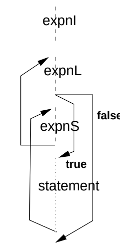
Similar to parse tree but only describes essential program structure.



Directed Acyclic Graphs

Used to represent control structure

for (expnI ; expnL ; expnS **)** statement



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?

Compiler Output Choices

- Assembly language (or other source code)
  Let existing assembler do some of the hard work.
  Makes code generation easier. Used in early C compilers.
- Relocatable machine code Usually an object module.
  Allows separate compilation, linking with existing libraries.
- Absolute machine code
  Generated code is directly executable.
- Interpretive pseudo code
  Machine instructions for some virtual machine. Used for portability and ease of compilation.
- High level programming language
  Example Specialized language $\longrightarrow$ C

Interpretive Systems

- Compiler generates a pseudo machine code that is a simple encoding of the program.
- The pseudo machine code is executed by another program (an *interpreter*)
- Interpreters are used for
  - 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 semantically analyzed statically.
  - Cases where run-time size must be minimized.
  - Implementing ugly language features.
  - Quick and dirty compilers.
  - As a way to port programs between environments.
- Interpreters lose on
  - Execution speed, usually significantly slower than machine code.
  - May limit user data space or size of programs.
  - May require recompilation for each run.

Examples of Interpreters

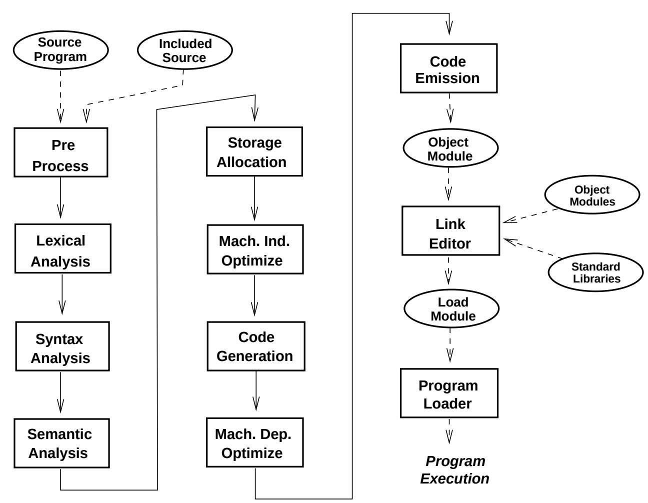
- Pascal P Machine
  - First compiler for Pascal compiled to a pseudo code (P-code) for a language-oriented stack machine.
  - Compiler for Pascal was provided in P-code and source.
  - Porting Pascal to new hardware only required writing a P-code interpreter for the new machine. 1..2 months work.
  - P-code influenced many later pseudo codes including U-code (optimization intermediate language) and Turing internal T-code.
- Java Virtual Machine[a]
  - Java programs are compiled to a *byte-code* for the *Java Virtual Machine* (JVM).
  - JVM designed to make Java portable to many platforms.
  - JVM slow execution speed has lead to the development of *Just In Time* (JIT) native code compilers for Java.

[a] See *Fischer, Cytron, LeBlanc* Section 10.2

Compiler Design Examples

- Student compiler
  - One pass for speed
  - In-memory tables
  - Compile to directly executable absolute code or to interpretive code.
  - Tune for compile speed and high quality diagnostics.
- Production Optimizing Compiler
  - Usually multi pass
  - Uses disk resident tables for large programs.
  - Data structures tuned for large programs.
  - Usually includes heavyweight optimization.

The Complete Compilation Process



Project Preview