CS 536

Introduction to Programming Languages and Compilers

Charles N. Fischer

Spring 2015

http://www.cs.wisc.edu/~fischer/cs536.html

Class Meets

Tuesdays, 5:30 — 8:30
Beatles Room, Epic Campus

Instructor

Charles N. Fischer
5393 Computer Sciences
Telephone: 608.262.1204
E- mail: fischer@cs.wisc.edu
Office Hours:
5:00 - 7:00, Monday & Thursday, Dune Room

Teaching Assistant

Menghui Wang
E- mail: menghui@cs.wisc.edu
Telephone: 608.262.1204
Office Hours:
To be determined

Key Dates

- February 10: Assignment #1 (Identifier Cross-Reference Analysis)
- March 3: Assignment #2 (CSX Scanner)
- March 24: Assignment #3 (CSX Parser)
- April 8: Midterm 1, 5:30 - 7:30 pm
- April 14: Assignment #4 (CSX Type Checker)
- April 15: Midterm 2, 5:00 - 7:00 pm
- May 6: Final Exam 1, 5:30 pm - 7:30 pm
- May 8: Assignment #5 (CSX Code Generator)
- May 12: Final Exam 2, 5:30 pm - 7:30 pm

Class Text

- **Crafting a Compiler**
  Fischer, Cytron, LeBlanc
  ISBN-10: 0136067050
  ISBN-13: 9780136067054
  Publisher: Addison-Wesley
- Handouts and Web-based reading will also be used.

Reading Assignment

- Chapters 1-2 of **CaC** (as background)

Class Notes

- The lecture notes used in each lecture will be made available prior to that lecture on the class Web page (under the “Lecture Nodes” link).

Piazza

Piazza is an interactive online platform used to share class-related information. We recommend you use it to ask questions and track course-related information. If you are enrolled (or on the waiting list) you should have already received an email invitation to participate (about one week ago).

Academic Misconduct Policy

- You must do your assignments—**no** copying or sharing of solutions.
- You may discuss general concepts and Ideas.
- All cases of Misconduct *must* be reported to the Dean’s office.
- Penalties may be **severe**.

Program & Homework Late Policy

- An assignment may be handed in up to one week late.
- Each late day will be debited 3%, up to a maximum of 21%.

Approximate Grade Weights

Program 1 - Cross-Reference Analysis 8%
Program 2 - Scanner 12%
Program 3 - Parser 12%
Program 4 - Type Checker 12%
Program 5 - Code Generator 12%
Homework #1 6%
Midterm Exam 19%
Final Exam (non-cumulative) 19%

Partnership Policy

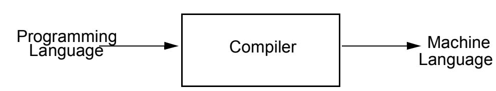
- Program #1 and the written homework must be done individually.
- Programs 2 to 5 may be done individually *or* by two person teams (your choice).

Compilers

Compilers are fundamental to modern computing.

They act as *translators*, transforming human- oriented *programming languages* into computer- oriented *machine languages.*

To most users, a compiler can be viewed as a “black box” that performs the transformation shown below.



A compiler allows programmers to ignore the machine- dependent details of programming.

Compilers allow programs and programming skills to be *machine- independent* and *platform- independent*.

Compilers also aid in detecting and correcting programming errors (which are all too common).

Compiler techniques also help to improve computer security. For example, the Java Bytecode Verifier helps to guarantee that Java security rules are satisfied.

Compilers currently help in protection of intellectual property (using *obfuscation*) and provenance (through *watermarking*).

Most modern processors are *multi- core* or *multi- threaded*. How can compilers find hidden parallelism in serial programming languages?

History of Compilers

The term *compiler* was coined in the early 1950s by Grace Murray Hopper. Translation was viewed as the "compilation" of a sequence of machine- language subprograms selected from a library.

One of the first real compilers was the FORTRAN compiler of the late 1950s. It allowed a programmer to use a problem-oriented source language.

Ambitious "optimizations" were used to produce efficient machine code, which was vital for early computers with quite limited capabilities.

Efficient use of machine resources is still an essential requirement for modern compilers.

Virtual Machine Code

Code generated by a compiler can consist *entirely* of virtual instructions (no native code at all). This allows code to run on a variety of computers.

Java, with its JVM (Java Virtual Machine) is a great example of this approach.

If the virtual machine is kept simple and clean, its interpreter can be easy to write. Machine interpretation slows execution by a factor of 3:1 to perhaps 10:1 over compiled code.

A "Just in Time" (*JIT*) compiler can translate "hot" portions of virtual code into native code to speed execution.

Advantages of Virtual Instructions

Virtual instructions serve a variety of purposes.

- They simplify a compiler by providing suitable primitives (such as method calls, string manipulation, and so on).
- They aid compiler transportability.
- They may decrease in the size of generated code since instructions are designed to match a particular programming language (for example, JVM code for Java).

Almost all compilers, to a greater or lesser extent, generate code for a virtual machine, some of whose operations must be interpreted.

The Structure of a Compiler

A compiler performs two major tasks:

- Analysis of the source program being compiled
- Synthesis of a target program

Almost all modern compilers are *syntax- directed:* The compilation process is driven by the syntactic structure of the source program.

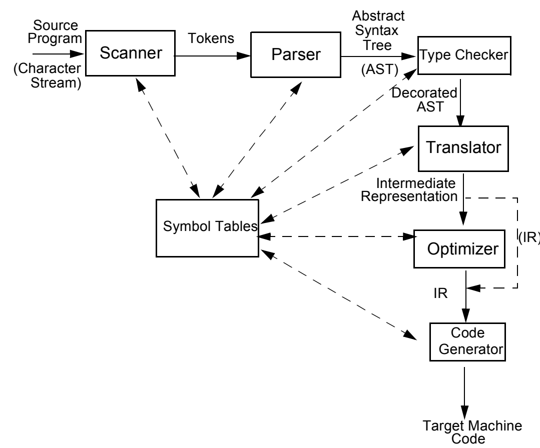
A parser builds semantic structure out of tokens, the elementary symbols of programming language syntax. Recognition of syntactic structure is a major part of the analysis task.

Semantic analysis examines the meaning (semantics) of the program. Semantic analysis plays a dual role.

It finishes the analysis task by performing a variety of correctness checks (for example, enforcing type and scope rules). Semantic analysis also begins the synthesis phase.

The synthesis phase may translate source programs into some intermediate representation (IR) or it may directly generate target code.

If an IR is generated, it then serves as input to a *code generator* component that produces the desired machine-language program. The IR may optionally be transformed by an *optimizer* so that a more efficient program may be generated.



The Structure of a Syntax-Directed Compiler

Scanner

The scanner reads the source program, character by character. It groups individual characters into tokens (identifiers, integers, reserved words, delimiters, and so on). When necessary, the actual character string comprising the token is also passed along for use by the semantic phases.

The scanner:

- Puts the program into a compact and uniform format (a stream of tokens).
- Eliminates unneeded information (such as comments).
- Sometimes enters preliminary information into symbol tables (for

example, to register the presence of a particular label or identifier).

- Optionally formats and lists the source program

Building tokens is driven by token descriptions defined using *regular expression* notation.

Regular expressions are a formal notation able to describe the tokens used in modern programming languages. Moreover, they can drive the *automatic generation* of working scanners given only a specification of the tokens. Scanner generators (like Lex, Flex and JLex) are valuable compiler- building tools.

Parser

Given a syntax specification (as a context- free grammar, CFG), the parser reads tokens and groups them into language structures.

Parsers are typically created from a CFG using a parser generator (like Yacc, Bison or Java CUP).

The parser verifies correct syntax and may issue a syntax error message.

As syntactic structure is recognized, the parser usually builds an abstract syntax tree (AST), a concise representation of program structure, which guides semantic processing.

Type Checker (Semantic Analysis)

The type checker checks the *static semantics* of each AST node. It verifies that the construct is legal and meaningful (that all identifiers involved are declared, that types are correct, and so on).

If the construct is semantically correct, the type checker “decorates” the AST node, adding type or symbol table information to it. If a semantic error is discovered, a suitable error message is issued.

Type checking is purely dependent on the semantic rules of the source language. It is independent of the compiler’s target machine.

Translator
(Program Synthesis)

If an AST node is semantically correct, it can be translated. Translation involves capturing the run- time "meaning" of a construct.

For example, an AST for a while loop contains two subtrees, one for the loop's control expression, and the other for the loop's body. *Nothing* in the AST shows that a while loop loops! This "meaning" is captured when a while loop's AST is translated. In the IR, the notion of testing the value of the loop control expression,

and conditionally executing the loop body becomes explicit.

The translator is dictated by the semantics of the source language. Little of the nature of the target machine need be made evident. Detailed information on the nature of the target machine (operations available, addressing, register characteristics, etc.) is reserved for the code generation phase.

In simple non- optimizing compilers (like our class project), the translator generates target code directly, without using an IR.

More elaborate compilers may first generate a high- level IR

(that is source language oriented) and then subsequently translate it into a low- level IR (that is target machine oriented). This approach allows a cleaner separation of source and target dependencies.

Optimizer

The IR code generated by the translator is analyzed and transformed into functionally equivalent but improved IR code by the optimizer.

The term optimization is misleading: we don't always produce the best possible translation of a program, even after optimization by the best of compilers.

Why?

Some optimizations are *impossible* to do in all circumstances because they involve an undecidable problem. Eliminating unreachable ("dead") code is, in general, impossible.

Other optimizations are too expensive to do in all cases. These involve NP- complete problems, believed to be inherently exponential. Assigning registers to variables is an example of an NP-complete problem.

Optimization can be complex; it may involve numerous subphases, which may need to be applied more than once.

Optimizations may be turned off to speed translation. Nonetheless, a well designed optimizer can significantly speed program execution by simplifying, moving or eliminating unneeded computations.

Code Generator

IR code produced by the translator is mapped into target machine code by the code generator. This phase uses detailed information about the target machine and includes machine- specific optimizations like *register allocation* and *code scheduling*.

Code generators can be quite complex since good target code requires consideration of many special cases.

Automatic generation of code generators is possible. The basic approach is to match a low- level IR to target instruction templates, choosing

instructions which best match each IR instruction.

A well- known compiler using automatic code generation techniques is the GNU C compiler. GCC is a heavily optimizing compiler with machine description files for over ten popular computer architectures, and at least two language front ends (C and C++).

Symbol Tables

A symbol table allows information to be associated with identifiers and shared among compiler phases. Each time an identifier is used, a symbol table provides access to the information collected about the identifier when its declaration was processed.

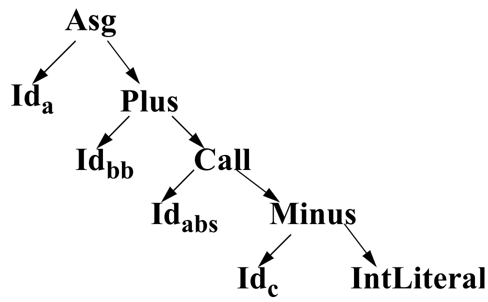
Example

Our source language will be *CSX*, a blend of C, C++ and Java.

Our target language will be the Java JVM, using the Jasmin assembler.

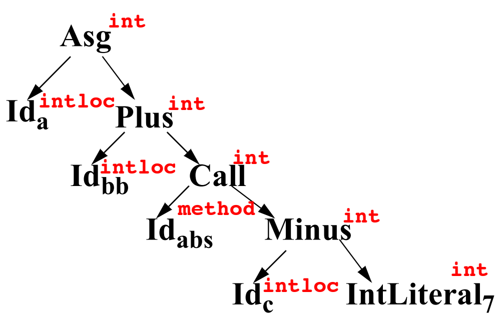
- A simple source line is
  `a = bb+abs(c-7);`
  this is a sequence of ASCII characters in a text file.
- The scanner groups characters into tokens, the basic units of a program.
  `a = bb+abs(c-7);`
  After scanning, we have the following token sequence:
  Id_a Asg Id_{bb} Plus Id_{abs} Lparen Id_c Minus $IntLiteral_7$ Rparen Semi

- The parser groups these tokens into language constructs (expressions, statements, declarations, etc.) represented in tree form:



(What happened to the parentheses and the semicolon?)

- The type checker resolves types and binds declarations within scopes:



- Finally, JVM code is generated for each node in the tree (leaves first, then roots):

```
iload  3  ; push local 3 (bb)
iload  2  ; push local 2 (c)
ldc    7  ; Push literal 7
isub      ; compute c-7
invokestatic  java/lang/Math/
abs(I)I
iadd      ; compute bb+abs(c-7)
istore  1 ; store result into
            local 1(a)
```

Symbol Tables & Scoping

Programming languages use scopes to limit the range in which an identifier is active (and visible).

Within a scope a name may be defined only once (though overloading may be allowed).

A symbol table (or dictionary) is commonly used to collect all the definitions that appear within a scope.

At the start of a scope, the symbol table is empty. At the end of a scope, all declarations within that scope are available within the symbol table.

A language definition may or may not allow *forward references* to an identifier.

If forward references are allowed, you may use a name that is defined later in the scope (Java does this for field and method declarations within a class).

If forward references are not allowed, an identifier is visible only after its declaration. C, C++ and Java do this for variable declarations.

In CSX no forward references are allowed.

In terms of symbol tables, forward references require two passes over a scope. First all

declarations are gathered. Next, all references are resolved using the complete set of declarations stored in the symbol table.

If forward references are disallowed, one pass through a scope suffices, processing declarations and uses of identifiers together.

Block Structured Languages

- Introduced by Algol 60, includes C, C++, CSX and Java.
- Identifiers may have a non- global scope. Declarations may be *local* to a class, subprogram or block.
- Scopes may *nest*, with declarations propagating to inner (contained) scopes.
- The lexically *nearest* declaration of an identifier is bound to uses of that identifier.

Example (drawn from C):

```
int x,z;
void A() {
  float x,y;
  print(x,y,z);
                        int
                      float
}                  float
void B() {
  print (x,y,z)
                         int
                           undeclared
}                      int
```


Block Structure Concepts

- Nested Visibility

  No access to identifiers outside their scope.

- Nearest Declaration Applies

  Using static nesting of scopes.

- Automatic Allocation and Deallocation of Locals

  Lifetime of data objects is bound to the scope of the Identifiers that denote them.

Is Case Significant?

In some languages (C, C++, Java and many others) case *is* significant in identifiers. This means **aa** and **AA** are different symbols that may have entirely different definitions.

In other languages (Pascal, Ada, Scheme, CSX) case *is not* significant. In such languages **aa** and **AA** are two alternative spellings of the same identifier.

Data structures commonly used to implement symbol tables usually treat different cases as different symbols. This is fine when case is significant in a language. When case is insignificant, you probably will

need to *strip case* before entering or looking up identifiers.

This just means that identifiers are converted to a uniform case before they are entered or looked up. Thus if we choose to use lower case uniformly, the identifiers **aaa**, **AAA**, and **AaA** are all converted to **aaa** for purposes of insertion or lookup.

BUT, inside the symbol table the identifier is stored in the form it was declared so that programmers see the form of identifier they expect in listings, error messages, etc.

How are Symbol Tables Implemented?

There are a number of data structures that can reasonably be used to implement a symbol table:

- An Ordered List
  Symbols are stored in a linked list, sorted by the symbol's name. This is simple, but may be a bit too slow if many identifiers appear in a scope.
- A Binary Search Tree
  Lookup is much faster than in linked lists, but rebalancing may be needed. (Entering identifiers in sorted order turns a search tree into a linked list.)
- Hash Tables
  The most popular choice.

Implementing Block-Structured Symbol Tables

To implement a block structured symbol table we need to be able to efficiently open and close individual scopes, and limit insertion to the innermost current scope.

This can be done using one symbol table structure if we tag individual entries with a "scope number."

It is far easier (but more wasteful of space) to allocate one symbol table for each scope. Open scopes are stacked, pushing and popping tables as scopes are opened and closed.

Be careful though—many preprogrammed stack implementations don't allow you to "peek" at entries below the stack top. This is necessary to lookup an identifier in all open scopes.

If a suitable stack implementation (with a peek operation) isn't available, a linked list of symbol tables will suffice.