CSC 488S/CSC 2107S Lecture Notes

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?

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.

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 $\rightarrow$ C

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.

In Conclusion: User Interfaces for Compilers

- The importance of good *Human Engineering* in designing the interface between the compiler and the user cannot be over emphasized.
- **The compiler should describe problems with a program in terms that the user can understand.**

Good: Syntax error on line 100 of file myProgram.c
The reserved word **if** cannot be followed by the identifier *myVar*

Bad: Illegal symbol pair **if** *identifier* . Parse stack dump:
123 < identifier>
122 **if**
121 < statement>
120 < block head>
...

Really Bad: Syntax error in program. Compilation terminated.

Unacceptable: java.lang.NullPointerException: null
or Segmentation fault - core dump

What the Compiler Should Do

- Determine if the program is correct.
- Describe the statically detectable errors at compile time.
- Translate the source program into an equivalent object program.
- Emit code to detect and describe dynamically detectable errors during program execution.
- Language designer (should) specify the possible errors in a language.
  Note the difference between error, implementation defined and unspecified.
- The language implementor decides between static and dynamic detection of errors.
- All errors *should be* detectable.

Error Detection Tradeoffs

- Static detection of errors may greatly increase
  - Complexity of the compilers internal data structures and algorithms
  - The time required to compile all programs.
- Example: To detect aliasing of variables the compiler needs links between all calls of a routine and the definition of the routine.
  The time to do aliasing detection is quadratic in the size of the program.
- Dynamic detection of errors at runtime may greatly increase
  - The size of the program due to extra chacking code (e.g. array subscript checking).
  - The execution time of the program due to the time required to execute checking code.
- Examples of expensive run time checking:
  - Uninitialized variable checking .
  - Array subscript checking .
  - Dangling and Nil pointer errors .

Compiler Reaction to Errors in Programs

Bad: Compiler Crash or Infinite Loop.

Bad: Java execption stack trace.

Bad: Generate incorrect object program and no error messages.

Poor: Stop the compilation

Good: Recover from the error and continue compilation.

Hard to do well. Error cascades and false errors are problems.

Great: Repair error and continue the compilation

Perfect correction is undecidable.

Use heuristics and systematic strategies.

Errors should be detected as soon as possible.

Localize (in a routine) the production of error messages.

Optionally print error summary at end of compilation

13 Errors detected, Last error on line 219 of file urProg.c

How the Compiler Can Help the User

- Use symbol table to produce information for the programmer
  - Cross reference list for identifier definition and use.
  - Frequency of usage information for identifiers.
  - Diagnostics for possible errors

    Variable/constant declared but never used.

    Variable assigned to but never used.
  - List sizes of data structures. Warn of excessive fill in data structures.
  - List code size and usage information for routines.
- Identify potentially inefficient constructs for the user.

  Statement 415 in yourProg.c generated 600 bytes of code.

  Statement 311 in utility.c implemented by 10 calls to library routines.
- Optionally summarize compiler internal resource usage.

  Used 400 of 511 symbol table entries.

  Provides warning of possible overrun of compiler limits.

- Provide user control over compiler processing options

  Example: `gcc` (see `man gcc` for the gory details).

  Good:

  - allows user control over optimization
  - provides work around for compiler problems

  Bad:

  - can become *very* complicated[a]

    For example gcc has

13 general options	19 C language options
3 language independ options	40 C++ language options
80 warning options	65 debugging options
131 optimization options	hundreds of machine dependent options

  - more than most users will ever use
  - makes software maintenance harder, since compiler options change compiled program

[a]Care to guess what the gcc option `-fmudflap` means?

Compiler Should be Self Diagnostic

- Compiler should be *error imune*, i.e. never crash for *any* possible input.
- Compiler must be programmed to detect *all* violations of compiler internal limits. Do internal consistency checking of compiler data structures.
- Use exception handlers to trap programming errors and shut down gracefully.

  An internal error has occurred. Please file a bug report.
- Optionally print out compiler internal performance statistics

  10,301 identifier lookups performed. 73% were local scope.

  4,219 scanner tokens produced, 3745 nodes in abstract syntax tree.

  143,216 bytes of code generated, 40,219 instructions.

  10.37 seconds in semantic analysis, 21.7 seconds in code generation.
- Optionally provide information (perhaps embedded in the object program) about the options used in the compilation

  Compiled on 2010-03-22 at 14:00:01 using superCompiler version 4.3.21

  Options -noCheckArray -O17 -stuffStructs -fastStrings

Example: `perl -V`

```
Summary of my perl5 (revision 5 version 14 subversion 2) configuration:
  Platform:
    osname=linux, osvers=2.6.42-37-generic, archname=x86_64-linux-gnu-thread-multi
    uname='linux batsu 2.6.42-37-generic #58-ubuntu smp thu jan 24 15:28:10 utc 20
    config_args='-Dusethreads -Duselargefiles -Dccflags=-DDEBIAN -Dcccdlflags=-fPI
    hint=recommended, useposix=true, d_sigaction=define
    useithreads=define, usemultiplicity=define
    useperlio=define, d_sfio=undef, uselargefiles=define, usesocks=undef
    use64bitint=define, use64bitall=define, uselongdouble=undef
    usemymalloc=n, bincompat5005=undef
  Compiler:
    cc='cc', ccflags ='-D_REENTRANT -D_GNU_SOURCE -DDEBIAN -fno-strict-aliasing -p
    optimize='-O2 -g',
    cppflags='-D_REENTRANT -D_GNU_SOURCE -DDEBIAN -fno-strict-aliasing -pipe -fsta
    ccversion='', gccversion='4.6.3', gccosandvers=''
    intsize=4, longsize=8, ptrsize=8, doublesize=8, byteorder=12345678
    d_longlong=define, longlongsize=8, d_longdbl=define, longdblsize=16
    ivtype='long', ivsize=8, nvtype='double', nvsize=8, Off_t='off_t', lseeksize=8
    alignbytes=8, prototype=define
  Linker and Libraries:
    ld='cc', ldflags =' -fstack-protector -L/usr/local/lib'
    libpth=/usr/local/lib /lib/x86_64-linux-gnu /lib/../lib /usr/lib/x86_64-linux-
    libs=-lgdbm -lgdbm_compat -ldb -ldl -lm -lpthread -lc -lcrypt
    perllibs=-ldl -lm -lpthread -lc -lcrypt
    libc=, so=so, useshrplib=true, libperl=libperl.so.5.14.2
    gnulibc_version='2.15'
```



```
  Dynamic Linking:
    dlsrc=dl_dlopen.xs, dlext=so, d_dlsymun=undef, ccdlflags='-Wl,-E'
    cccdlflags='-fPIC', lddlflags='-shared -O2 -g -L/usr/local/lib -fstack-protect
Characteristics of this binary (from libperl):
  Compile-time options: MULTIPLICITY PERL_DONT_CREATE_GVSV
                        PERL_IMPLICIT_CONTEXT PERL_MALLOC_WRAP
                        PERL_PRESERVE_IVUV USE_64_BIT_ALL USE_64_BIT_INT
                        USE_ITHREADS USE_LARGE_FILES USE_PERLIO USE_PERL_ATOF
                        USE_REENTRANT_API
  Locally applied patches:
    <DELETED for brevity>
  Built under linux
  Compiled at Mar 18 2013 19:17:55
  @INC:
    /etc/perl
    /usr/local/lib/perl/5.14.2
    /usr/local/share/perl/5.14.2
    /usr/lib/perl5
    /usr/share/perl5
    /usr/lib/perl/5.14
    /usr/share/perl/5.14
    /usr/local/lib/site_perl
```