CS 375: Compilers

Fall 2008

Course website: http://www.cs.utexas.edu/~pingali/CS375/2008fa/

Course staff

- Instructor:
  - Keshav Pingali
  - Department of Computer Science and ICES
  - Office: ACES 4.126
  - email: pingali@cs.utexas.edu
- TA:
  - Donald Nguyen
  - Department of Computer Sciences
  - Office: ACES 4.104
  - email: ddn@cs.utexas.edu

Meeting times

- Lecture:
  - TR 12:30-2:00 PM
  - Location: RAS 310
- My office hours:
  - Thursday 3:00-4:00 PM
- Donald's office hours:
  - Wednesday 10am-11am
  - ENS 31NQ Desk #3

Textbook

- "Compilers" by Aho, Lam, Sethi, Ullman
  - Addison-Wesley Publishers
- Text-book is required
  - you may get homework assignments from this book
  - no assigned readings per se, since I will assume you are smart enough to figure out what to read
- Optional books:
  - "Engineering a compiler" by Torczon and Cooper (2003)

Online resources

- Most top CS departments have good compiler courses
- Lecture notes and course material are usually online
- I will use material from CS 412 at Cornell
  - http://www.cs.cornell.edu/courses/cs412/
  - Lecture notes from Radu Rugina

Course work

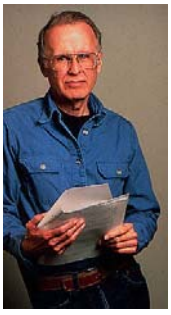
- First month: 3 programming assignments + homework problems
- Rest of semester: project, broken up into 3-4 assignments
- All assignments should be done in groups of two students
- No exams

What is a compiler?

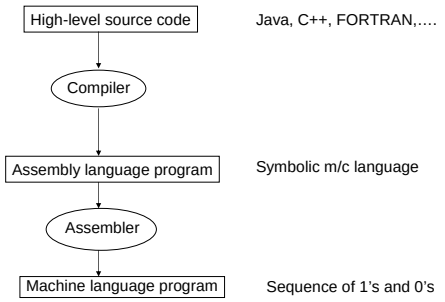
- Program to translate information from one representation to another
  - usually the information is another program
  - usually the translation is from high-level (human-oriented) language to low-level (machine-oriented) language
- Examples: gcc, javac, IBM xlf/xlc, ….
- Atypical examples:
  - Latex: compiles a LaTeX document → dvi
  - IC compilers: high-level circuit specs → chips

Origins of compilers

- In the early days of digital computers, programs were written in machine language or assembly language
- Not a very effective use of people
- 1957: FORTRAN I language and compiler
  - John Backus and team at IBM
  - Turing award (1977)
  - National Medal of Science (1975)



High-level Structure of Compiler



Assembly/machine language

- Asssembly/machine language of most real machines are quite complex
- To avoid introducing these complications right away, we will start with
  - SaM: simple stack machine for teaching compilers
  - you can download a SaM simulator (jar file) from the course homepage and play with it
- SaM is modeled vaguely after the Java Virtual Machine (JVM) but it is simpler

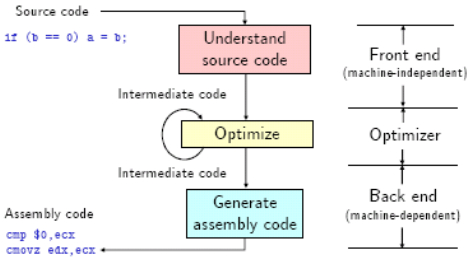
Factorial: SaM assembly code

```
main: PUSHIMM 0 //return value slot for main
      //set up call to fact
      PUSHIMM 0//return value slot for abs
      PUSHIMM 5//parameter to abs
      LINK//save FBR and update FBR
      JSR fact//call fact
      POPFBR //restore FBR
      ADDSP –1//pop off parameter
      //end of call to abs
       JUMP mainEnd
mainEnd:
      STOREOFF 0//store result of call
          //into return value slot of main
      STOP
fact: PUSHOFF –1//get n
      ISNIL //is it zero
      JUMPC zer//if so, jump to zer
      PUSHOFF –1//get n
      //set up recursive call
      PUSHIMM 0//rv slot
      PUSHOFF –1//compute n-1
      PUSHIMM 1
      SUB
      LINK//save FBR
      JSR fact//call fact recursively
      POPFBR//restore FBR
      ADDSP –1//pop parameter
      TIMES//n*fact(n-1)
      JUMP factEnd
zer:  PUSHIMM 1//push 1
      JUMP factEnd
factEnd: STOREOFF –2//store into r.v.
      JUMPIND//return
```

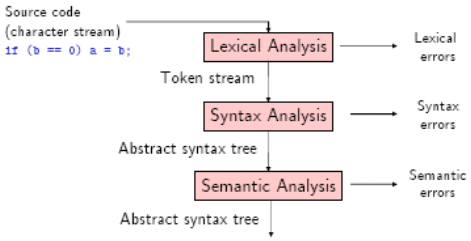
Program optimization

- If you study the SaM code on the previous slide, you will find a few places where you can eliminate instructions without changing the output of the program
  - program optimization
- SaM code for factorial was generated by a naïve compiler that did not do optimizations
- Modern compilers are *optimizing* compilers

Optimizing compiler structure



Front-end structure



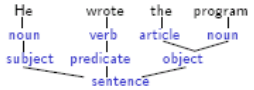
Syntax analysis is also known as parsing.

Analogy

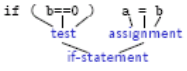
- Front end can be explained by analogy to the way humans understand natural languages
- Lexical analysis
  - Natural language: "He wrote the program"
    words: "he" "wrote" "the" "program"
  - Programming language `"if (b == 0) a = b"`
    tokens: `"if"` `"("` `"b"` `"=="` `"0"` `")"` `"a"` `"="` `"b"`

Analogy (contd.)

- Syntactic analysis
  - Natural language:



  - Programming language



Analogy (contd.)

- Semantic analysis
  - Natural language:

    He wrote the computer
    noun verb article noun

    Syntax is correct; semantics is wrong!

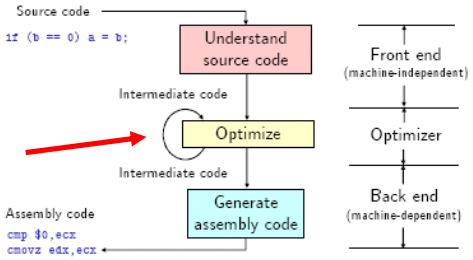
  - Programming language

    `if ( b == 0 ) a = foo`
    test assignment

    if `a` is an integer and `foo` is a method, the compiler will complain.

One important kind of semantic analysis is type checking.

Compiler Structure



Optimization

- Goal:
  - convert intermediate language program into a more efficient intermediate language program without changing the meaning of the program
- Examples of optimizations:
  - Constant folding
    - evaluate expressions at compile-time (2+3 → 5)
    - if this expression occurs inside a loop, we would evaluate it just once at compile-time rather than each time through the loop at runtime
  - Loop-invariant removal
    - compute expressions once outside a loop rather than in each iteration, if possible
  - Common sub-expression elimination
    - avoid re-computing expressions if the values of operands have not changed
  - ….
- Two phases:
  - Analysis: determine opportunities for optimizing program safely
  - Transformation: rewrite the program
- Question:
  - Do we need optimizing compilers only because we have dumb programmers?
  - Hint: can you do all optimizations at the source level?

Assembly code generation

- Register allocation
  - Which values should reside in registers and which values should reside in memory?
- Instruction selection
  - What sequence of assembly language instructions should be used to implement each intermediate code statement?

Conventional compiler courses

- Most compiler courses start with lexical analysis and slog their way through to code generation over the course of the semester.
- You do not see the big picture till the very end of the course.
- You can even get lost in the details and never see the big picture….

CS 375 Experiment

- Two passes over material
- First pass:
  - roughly 5-6 lectures
  - learn enough to implement a quick and dirty compiler for C-like language to SaM code
    - no type checking or optimizations
    - lexical analyzer will be given to you
    - parser: recursive-descent
- Second pass:
  - rest of the course
  - learn
    - theory of grammars and parsing
    - type checking
    - optimizations
    - register allocation
    - ….
  - implement a more ambitious compiler for an OO-language to a real machine (we are still trying to figure out what to do here)