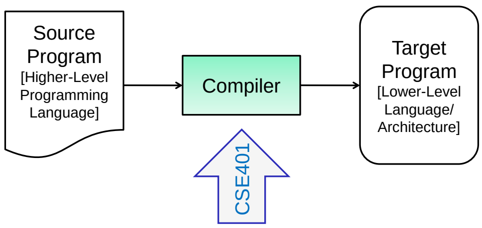


Introduction to Compiler Construction

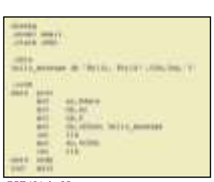
David Notkin
Autumn 2008

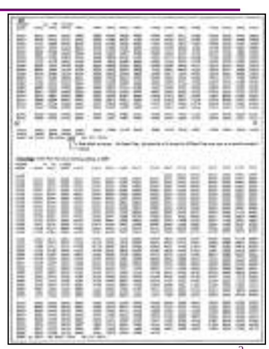
"Compiler": from the web

- The Oxford English Dictionary (OED) indicates that the first usage of the term is circa 1330, referring to one who collects and puts together materials
  - They also note a usage "Diuerse translatours and compilaris" from Scotland in 1549
- Most dictionaries give the above definition as well as the computing-based definition (which the OED dates to 1953)
  - A program that translates programs written in a high-level programming language into equivalent programs in a lower-level language
- Wikipedia credits Grace Hopper with the first compiler (for a language called A-0) in 1952, and John Backus' IBM team with the first complete compiler (for FORTRAN) in 1957

Trivia: In what year was I born?

A world with no compilers





Assembly/machine language coding

- …is slow, error-prone, tedious, not portable, …
- The size (roughly, lines of code) of a high-level language program relative to its assembly language equivalent is approximately linear – but that may well be a factor of 10 or even 100
  - Microsoft Vista is something like 50 million lines of source code (50 MLOC)
    - Printed double-sided something like triple the height of the Allen Center
    - Something like 20 person-years *just to retype*
- Q: Why is harder to build a program 10 times larger?

Ergo: we need compilers

- And to have compilers, somebody has to build compilers
  - At least every time there is a need to program in a new <programming language, architecture> pair
  - Roughly how many pl's and how many ISA's? Cross product?
- Unless the compilers could be generated automatically – and parts can (a bit more on this later in the course)

Trivia: In what year did I first write a program? In what language? On what architecture?

CSE401 Au08 5

But why might you care?

- Crass reasons: jobs
- Class reasons: grade in 401
- Cool reasons: loveliest blending of theory and practice in computer science & engineering
- Cruel reasons: we all had to learn it ☺
- Practice reasons: more experience with software design, modifying software written by others, etc.
- Practical reasons: the techniques are widely used outside of conventional compilers
- Super-practical reasons: lays foundation for understanding or even researching really cool stuff like JIT (just-in-time) compilers, compiling for multicore, building interpreters, scripting languages, (de)serializing data for distribution, and more…

CSE401 Au08 6

Better understand…

- Compile-time vs. run-time
- Interactions among
  - language features
  - implementation efficiency
  - compiler complexity
  - architectural features

CSE401 Au08 7

Compiling (or related) Turing Awards

- 1966 Alan Perlis
- 1972 Edsger Dijkstra
- 1976 Michael Rabin and Dana Scott
- 1977 John Backus
- 1978 Bob Floyd
- 1979 Bob Iverson
- 1980 Tony Hoare
- 1984 Niklaus Wirth
- 1987 John Cocke
- 2001 Ole-Johan Dahl and Kristen Nygaard
- 2003 Alan Kay
- 2005 Peter Naur
- 2006 Fran Allen

CSE401 Au08 8

Questions?

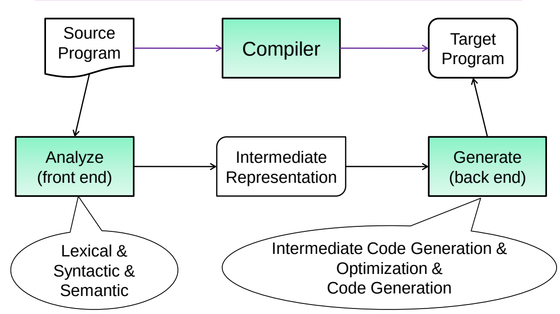
Administrivia: see web

- Text: Engineering a Compiler, Cooper and Torczon, Morgan-Kaufmann 2004
- Mail list – automatically subscribed
- Google calendar with links
- Grading
  - Project 40%
  - Homework 15%
  - Midterm 15%
  - Final 25%
  - Other (class participation, extra credit, etc.) 5%

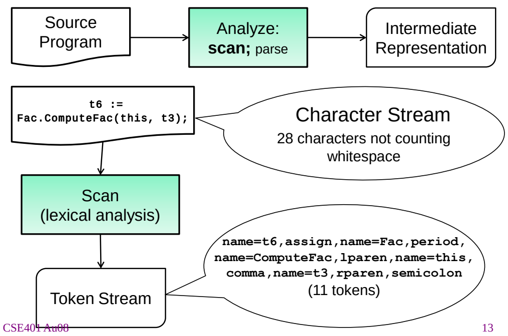
Project

- Start with a MiniJava compiler in Java
- Add features such as comments, floating-point, arrays, class variables, for loops, etc.
- Completed in stages over the term
- Not teams: but you can talk to each other (“Prison Break” rule, see web) for the project
- Grading basis: correctness, clarity of design and implementation, quality of test cases, etc.

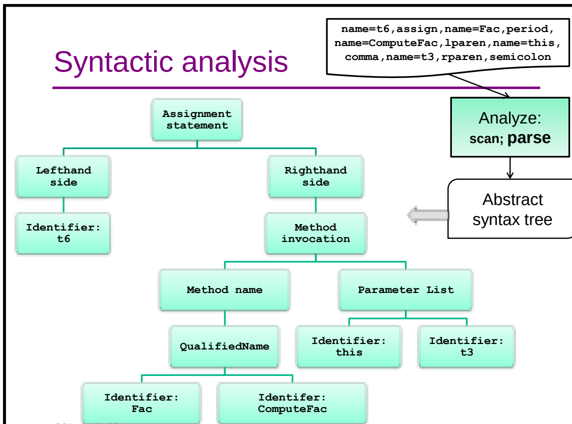
Compiler structure: overview



Lexical analysis (scanning, lexing)

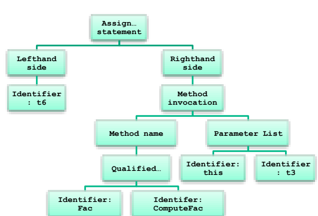


Syntactic analysis

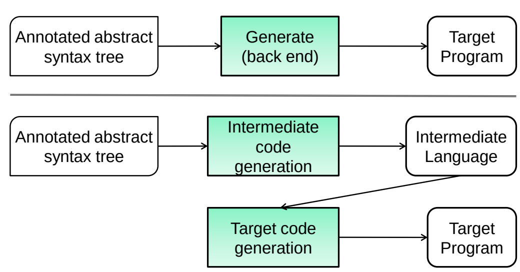


Semantic analysis

- Annotate abstract syntax tree
- Primarily determine which identifiers are associated with which declarations
- Scoping is key issue
- Symbol table is key data structure



Code generation (backend)



Optimization

- Takes place at various (and multiple) places during code generation
  – Might optimize the intermediate language code
  – Might optimize the target code
  – Might optimize during execution of the program
- Q: Is it better to have an optimizing compiler or to hand-optimize code?

Quotations about optimization

- Michael Jackson
  – Rule 1: Don't do it.
  – Rule 2 (for experts only): Don't do it yet.
- Bill Wulf
  – More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason – including blind stupidity.
- Don Knuth
  – We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.

Questions?

Lexing: reprise

- Read in characters
- Clump into tokens
- Strip out whitespace and comments
- Tokens are specified using regular expressions

```
Ident ::= Letter AlphaNum*
Integer ::= Digit+
AlphaNum ::= Letter | Digit
Letter ::= 'a' | … | 'z' | 'A' | … | 'Z'
Digit ::= '0' | … | '9'
```

- Q: regular expressions are equivalent to something you've previously learned about… what is it?

Syntactic analysis: reprise

- Read in tokens
- Build a tree based on syntactic structure
- Report any syntax errors
- EBNF (extended Backus-Naur Form) is a common notation for defining programming language syntax as a context-free grammar

```
Stmt ::= if (Expr) Stmt [else Stmt]
   | while (Expr) Stmt | ID = Expr; | …
Expr ::= Expr + Expr | Expr < Expr | … | ! Expr
   | Expr . ID ([Expr {,Expr}])
   | ID | Integer | (Expr) | …
```

- The grammar specifies the concrete syntax of language
- The parser constructs the abstract syntax tree

Semantic analysis: reprise

- Do name resolution and type checking on the abstract syntax tree
  - What declaration does each name refer to?
  - Are types consistent? Are other static properties consistent?
- Symbol table
  - maps names to information about name derived from declaration
  - represents scoping usually through a tree of per-scope symbol tables
- Overall process
  1. Process each scope top down
  2. Process declarations in each scope into symbol table
  3. Process body of each scope in context of symbol table

Intermediate code generation: reprise

- Translate annotated AST and symbol tables into lower-level intermediate code
- Intermediate code is a separate language
  - Source-language independent
  - Target-machine independent
- Intermediate code is simple and regular
  - Good representation for doing optimizations
  - Might be a reasonable target language itself, e.g. Java bytecode

Target code generation: reprise

- Instruction selection: choose target instructions for (subsequences) of intermediate representation (IR) instructions
- Register allocation: allocate IR code variables to registers, spilling to memory when necessary
- Compute layout of each procedures stack frames and other runtime data structures
- Emit target code

Example: source

```
Sample (extended) MiniJava program: Factorial.java
// Computes 10! and prints it out
class Factorial {
    public static void main(String[] a) {
         System.out.println(
               new Fac().ComputeFac(10));
    }
}
class Fac {
   // the recursive helper function
   public int ComputeFac(int num) {
      int numAux;
      if (num < 1)
         numAux = 1;
      else numAux = num * this.ComputeFac(num-1);
      return numAux;
   }
}
```


Example: intermediate representation

```
Int Fac.ComputeFac(*? this, int num) {
  int t1, numAux, t8, t3, t7, t2, t6, t0;
  t0 := 1;
  t1 := num < t0;
  ifnonzero t1 goto L0;
  t2 := 1;
  t3 := num - t2;
  t6 := Fac.ComputeFac(this, t3);
  t7 := num * t6;
  numAux := t7;
  goto L2;
label L0;
  t8 := 1;
  numAux := t8
label L2;
  return numAux
}
```


Questions?

Don’t forget

- Survey (before Friday)
- Readings (on calendar)
- Visit office hours (on calendar)
- Ask questions