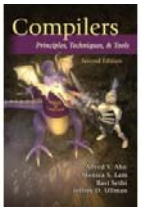
COMP 181
Compilers

Lecture 2
Overview

September 7, 2006

Administrative

- Book?
  - Hopefully:
    "*Compilers*" by Aho, Lam, Sethi, Ullman
- Mailing list
- Handouts?
- Programming assignments
  For next time, write a "hello, world" program in Java



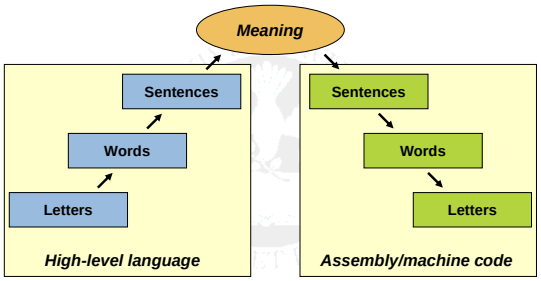
Prelude

- What type of plane is this?
  Boeing 777
- What is notable about the design and construction of 777?
  Done completely on computer
- Today's lecture: the view from 35,000 feet

Last time…

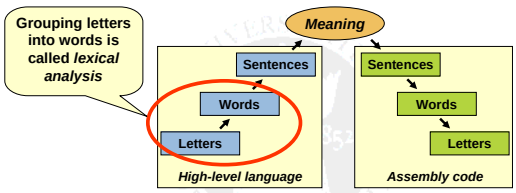
- The compilation problem
  - Source language
    - High-level abstractions
    - Easy to understand and maintain
  - Target language
    - Very low-level, close to machine
    - Few abstractions
- Concerns
  - Systematic, correct translation
  - High-quality translation

Translation strategy



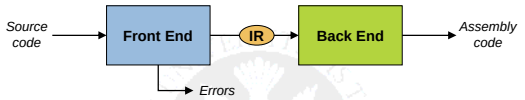
Compilation strategy

- Follows directly from translation strategy:



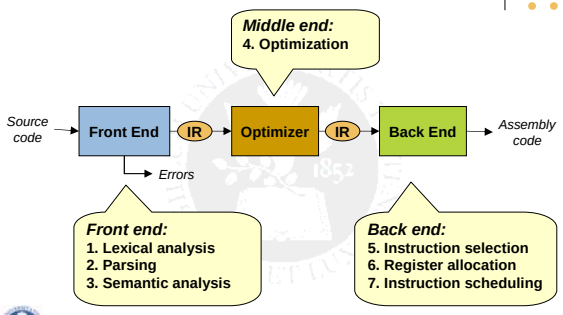
- A series of ***passes***
  - Each pass performs one step
  - Transforms the program representation

Basic compiler structure

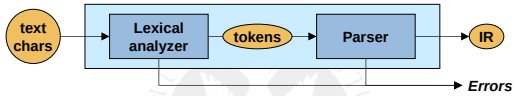


- Traditional two-pass compiler
  - ***Front-end*** reads in source code, checks for errors
  - ***Internal representation*** captures meaning
  - ***Back-end*** generates assembly
- Advantage?
  - Decouples input language from target machine

Modern optimizing compiler



The front end



- Responsibilities
  - Recognize legal (and illegal programs)
  - Report errors in a useful way
  - Generate internal representation
- How it works
  - ***Good news***: linear time, mostly generated automatically
  - By analogy to natural languages…

Lexical Analysis

- First step: recognize words.
  - Smallest unit above letters

This is a sentence.

- Note the
  - Capital "T" (start of sentence symbol)
  - Blank " " (word separator)
  - Period "." (end of sentence symbol)

More Lexical Analysis

- Lexical analysis is not trivial. Consider:

  ist his ase nte nce

- Plus, programming languages are typically more cryptic than English:

  *p->f ++ = -.12345e-5

Lexical analysis

- Another example:

```
void func(float * p, float y)
{
  float x;
  x = y/*p;
}
```

- What happens in this case?

  "`/*`" is the comment delimiter

And More Lexical Analysis

- Lexical analyzer divides program text into "words" or ***tokens***

  if x == y then z = 1; else z = 2;

- Tokens have value and type:

  <if, *keyword*>, <x, *identifier*>, <==, *operator*>, etc….

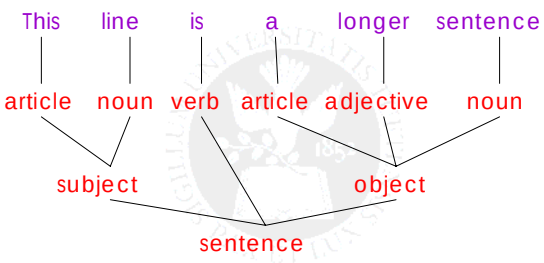
Specification

- How do we specify tokens?
  - Keyword – an exact string
  - What about identifier? floating point number?
- Regular expressions
  - Just like Unix tools grep, awk, sed, etc.
  - Identifier: `[a-zA-Z_][a-zA-Z_0-9]*`
  - Algorithms for matching regexps
    - Actually, generate code that does the matching
    - This code is often called a ***scanner***

Parsing

- Once words are understood, the next step is to understand sentence structure
- Parsing = Diagramming Sentences
  - The diagram is a tree…

Diagramming a Sentence

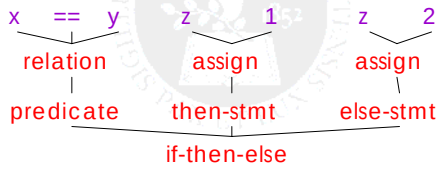


Diagramming programs

- Diagramming program expressions is the same
- Consider:

  If x == y then z = 1; else z = 2;

- Diagrammed:



Specification

- How do we describe the language?

  Same as English: using grammar rules

1. *sentence* → *subject verb object*
2. *subject* → *noun-phrase*
3. *noun-phrase* → *article noun-phrase*
4. | *adjective noun-phrase*
5. | *noun*

…etc…

1. *goal* → *expr*
2. *expr* → *expr op term*
3. | *term*
4. *term* → number
5. | id
6. *op* → +
7. | -

Tokens from scanner

- Formal grammars
  - Chomsky hierarchy – ***context-free grammars***
  - Each rule is called a ***production***

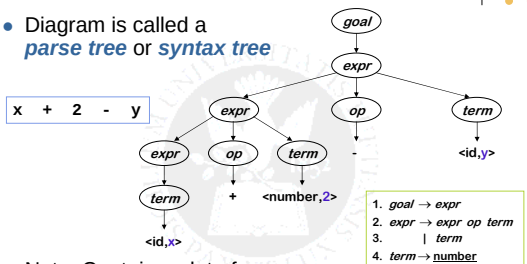
Using grammars

- Given a grammar, we can ***derive*** sentences by repeated substitution
- ***Parsing*** is the reverse process – given a sentence, find a derivation (same as diagramming)

Production	Result
	goal
1	*expr*
2	*expr op term*
5	*expr op* y
7	*expr* - y
2	*expr op term* - y
4	*expr op* 2 - y
6	*expr* + 2 - y
3	*term* + 2 - y
5	x + 2 - y

Representation

- Diagram is called a ***parse tree*** or ***syntax tree***

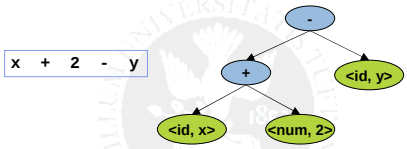


1. *goal* → *expr*
2. *expr* → *expr op term*
3. | *term*
4. *term* → number
5. | id
6. *op* → +
7. | -

- Note: Contains a lot of unneeded information.

Representation

- Compilers often use an ***abstract syntax tree***



- More concise and convenient:
  - Summarizes grammatical structure without including all the details of the derivation
  - ASTs are one kind of *intermediate representation (IR)*

Semantic Analysis

- Once sentence structure is understood, we can try to understand "meaning"
  - But meaning is too hard for compilers
- Compilers perform limited analysis to catch inconsistencies
- Some do more analysis to improve the performance of the program

Semantic Analysis in English

- Example:

  Jack said Jerry left his assignment at home.

  What does "his" refer to? Jack or Jerry?
- Even worse:

  Jack said Jack left his assignment at home?

  How many Jacks are there?

  Which one left the assignment?

Semantic analysis in programs

- Programming languages define strict rules to avoid such ambiguities
- This Java code prints "4"; the inner definition is used

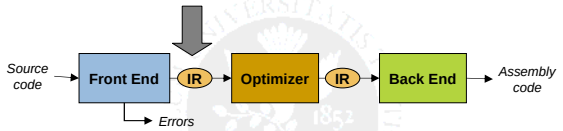
```
{
    int Jack = 3;
    {
        int Jack = 4;
        System.out.
            print(Jack);
    }
}
```

More Semantic Analysis

- Compilers perform many semantic checks besides variable bindings
- Example:

  Jack left her homework at home.
- A "type mismatch" between her and Jack; we know they are different people
  - Presumably Jack is male

Where are we?



- Front end
  - Produces fully-checked AST
  - Problem: AST still represents source-level semantics

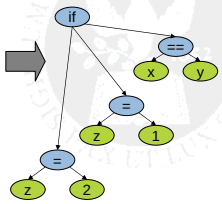
Intermediate representations

- Many different kinds of IRs
  - High-level IR (e.g. AST)
    - Closer to source code
    - Hides implementation details
  - Low-level IR
    - Closer to the machine
    - Exposes details (registers, instructions, etc)
  - Many tradeoffs in IR design
- Most compilers have 1 IR:
  - Typically closer to low-level IR
  - Better for optimization and code generation

IR lowering

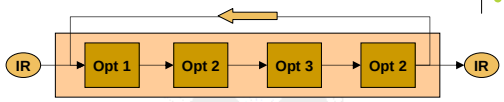
- Preparing for optimization and code gen
  - Dismantle complex structures into simple ones
  - Process is called ***lowering***
  - Result is an IR called ***three-address code***

```
if (x == y)
  z = 1;
else
  z = 2;
```




```
t0 = x == y
br t0 label1
goto label2
label1:
z = 1
goto label3
label2:
z = 2
label3:
```


Optimization



- Series of passes – often repeated
  - ***Goal***: reduce some cost
    - Run faster
    - Use less memory
    - Conserve some other resource, like power
  - Must preserve program semantics
- Dominant cost in most modern compilers

Optimization

- General scheme
  - Analysis phase:
    - Pass over code looking for opportunities
    - Often uses a formal analysis framework
  - Transformation phase
    - Modify the code to exploit opportunity
- Classic optimizations
  - Dead-code elimination, common sub-expression elimination, loop-invariant code motion, strength reduction
- This class: time permitting

Optimization example

- Array accesses

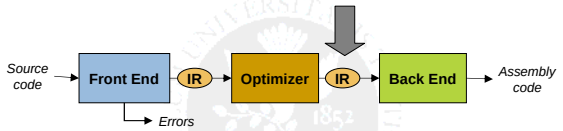
```
for (i = 0; i < N; i++)
  for (j = 0; j < M; j++)
    A[i][j] = A[i][j] + C;
```

```
for (i = 0; i < N; i++)
  for (j = 0; j < M; j++){
    t0 = &A + (i * M) + j
    (*t0) += C;
  }
```

```
for (i = 0; i < N; i++) {
  t1 = i * M
  for (j = 0; j < M; j++){
    t0 = &A + t1 + j
    (*t0) += C;
  }
}
```

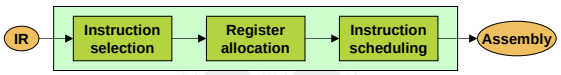
```
t1 = 0;
for (i = 0; i < N; i++) {
  t1 = t1 + M;
  for (j = 0; j < M; j++){
    t0 = &A + t1 + j
    (*t0) += C;
  }
}
```


Where are we?



- Optimization output
  - Transformed program
  - Typically, same level of abstraction

Back end



- Responsibilities
  - Map abstract instructions to real machine architecture
  - Allocate storage for variables in registers
  - Schedule instructions (often to exploit parallelism)
- How it works
  - ***Bad news***: very expensive, poorly understood, little automation

Instruction selection

- Example: RISC instructions

```
...
label1:
t1 = b * c
y = a + t1
z = d + t1
...
```

```
load @b       => r1
load @c       => r2
mult r1, r2   => r3
load @a       => r4
add r3, r4    => r5
store r5      => @y
load @d       => r6
add r3, r6    => r7
store r7      => @z
```

- Notice:
  - Explicit loads and stores
  - Lots of registers – *"virtual registers"*

Register allocation

- Goals:
  - Have each value in a register when it is used
  - Manage a limited set of res
  - Often need to insert loads a

Pentium 4	
Registers	1 cycle
Cache	3-8 cycles
Memory	30-150 cycles

- Algorithms
  - Optimal allocation is NP-complete
  - Many back-end algorithms compute approximate solutions to NP-complete problems

Instruction scheduling

- Conflicting goals:
  - Move loads early to avoid waiting
  - Move operations together to reduce registers

```
load @b       => r1
load @c       => r2
mult r1, r2   => r3
load @a       => r1
add r3, r1    => r1
store r1      => @y
load @d       => r1
add r3, r1    => r1
store r1      => @z
```

Uses only 3 registers, but may stall on loads

```
load @b       => r1
load @c       => r2
load @a       => r4
load @d       => r5
mult r1, r2   => r3
add r3, r4    => r4
store r4      => @y
add r3, r5    => r5
store r5      => @z
```

Start loads early, hide latency, but need 5 registers

Finished program

- What else does the code need to run?
- Programs need support at run-time
  - Start-up code
  - Interface to OS
  - Libraries
- Varies significantly between languages
  - C – fairly minimal
  - Java – Java virtual machine

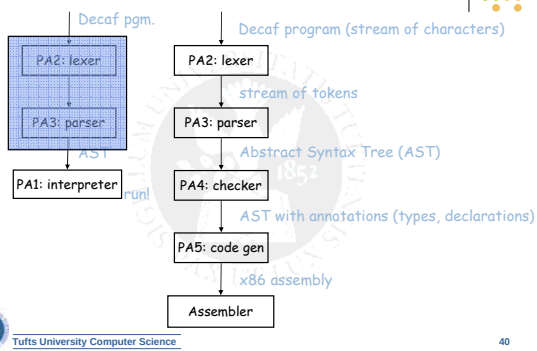
Run-time System

- Memory management services
  - Manage heap allocation
  - Garbage collection
- Run-time type checking
- Error processing (exception handling)
- Interface to the operating system
- Support of parallelism
  - Parallel thread initiation
  - Communication and synchronization

Programming Assignments

- The project
  - A compiler for Decaf, a subset of Java
  - Generates x86 code (as opposed to bytecodes)
  - Implemented in Java
- Five parts:
  - PA1: interpreter of a subset of Decaf
  - PA2-5: the compiler of Decaf, in four pieces
    - PA2: lexical analysis (a.k.a. scanner, lexer)
    - PA3: syntactic analysis (a.k.a. parser)
    - PA4: semantic analyzer (a.k.a. type checker)
    - PA5: code generator

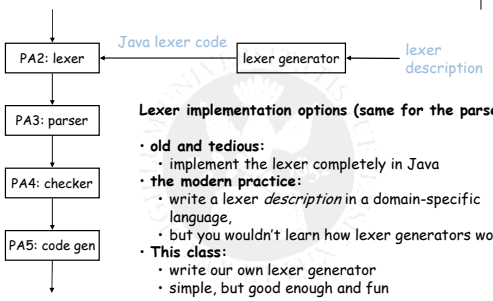
The Decaf compiler



Next time…

- Lexical analysis
- First programming assignment

How we will implement the scanner, parser



Lexer implementation options (same for the parser):

- **old and tedious:**
  - implement the lexer completely in Java
- **the modern practice:**
  - write a lexer *description* in a domain-specific language,
  - but you wouldn't learn how lexer generators work
- **This class:**
  - write our own lexer generator
  - simple, but good enough and fun