Intro to Compilers

CSE 413, Autumn 2007
11-07-2007

Agenda

- What's a compiler?
- Compilers vs. Interpreters
- Phases of a compiler

And the point is…

- Execute this!

```
int nPos = 0;
int k = 0;
while (k < length) {
  if (a[k] > 0) {
    nPos++;
  }
}
```

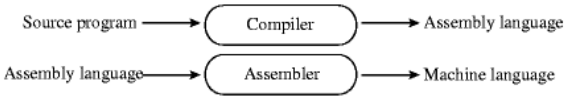
- How?

Compiler

- Read and analyze entire program
- Translate to semantically equivalent program in another language
  - Presumably easier to execute or more efficient
  - Should "improve" the program in some fashion
- Offline process
  - Tradeoff: compile time overhead (preprocessing step) vs execution performance

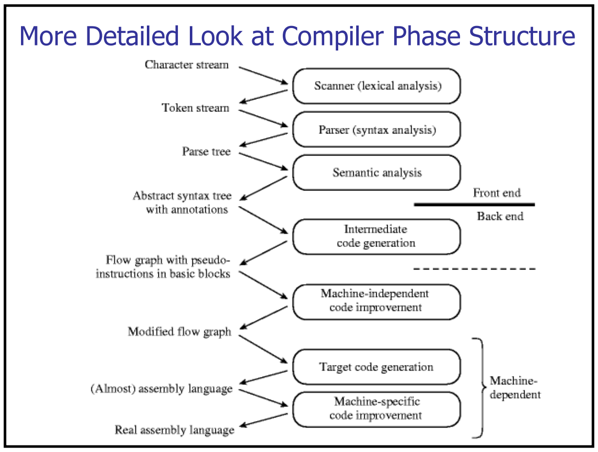
---

Compiler vs. Assembler



Assembler

- Principal tasks of an assembler are:
  - Replace opcodes and operands with their machine language encodings
  - Replace uses of symbolic names with actual addresses
- Assembler translates assembly language into Object code (Machine code)



More Detailed Look at Compiler Phase Structure

Character stream

Scanner (lexical analysis)

Token stream

Parser (syntax analysis)

Parse tree

Semantic analysis

Abstract syntax tree
with annotations

Front end

Back end

Intermediate
code generation

Flow graph with pseudo-
instructions in basic blocks

Machine-independent
code improvement

Modified flow graph

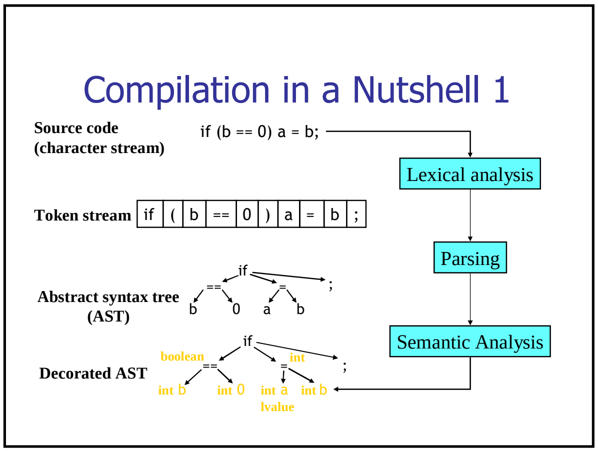
Target code generation

(Almost) assembly language

Machine-specific
code improvement

Real assembly language

Machine-
dependent



Compilation in a Nutshell 1

Source code
(character stream)

if (b == 0) a = b;

Lexical analysis

Token stream

if	(	b	==	0	)	a	=	b	;

Parsing

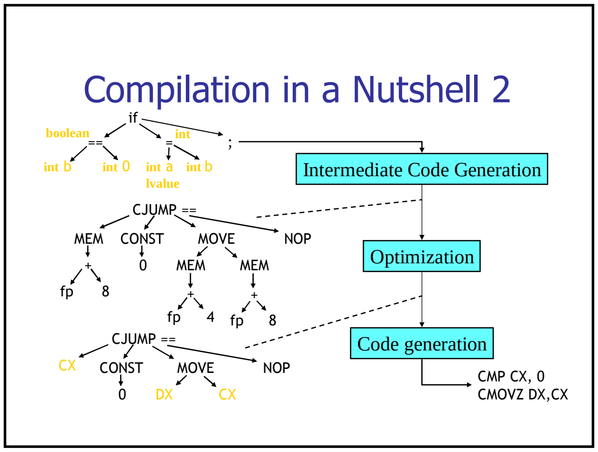
Abstract syntax tree
(AST)

if
==
b
0
=
a
b
;

Semantic Analysis

Decorated AST

if
boolean
==
int b
int 0
int
=
int a
lvalue
int b
;



Compilation in a Nutshell 2

if
boolean
==
int b
int 0
int
=
int a
lvalue
int b
;

Intermediate Code Generation

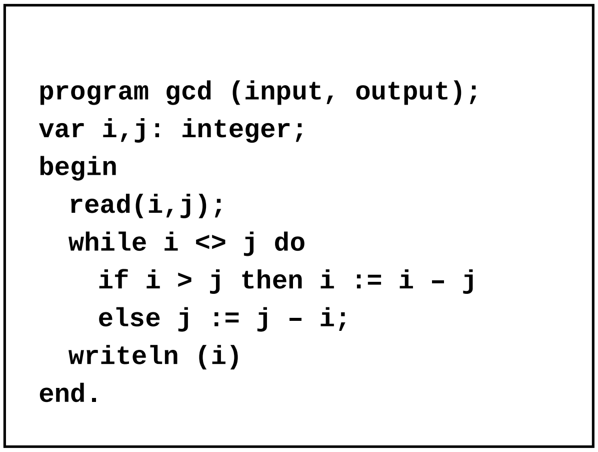
CJUMP ==
MEM
+
fp
8
CONST
0
MOVE
MEM
+
fp
4
MEM
+
fp
8
NOP

Optimization

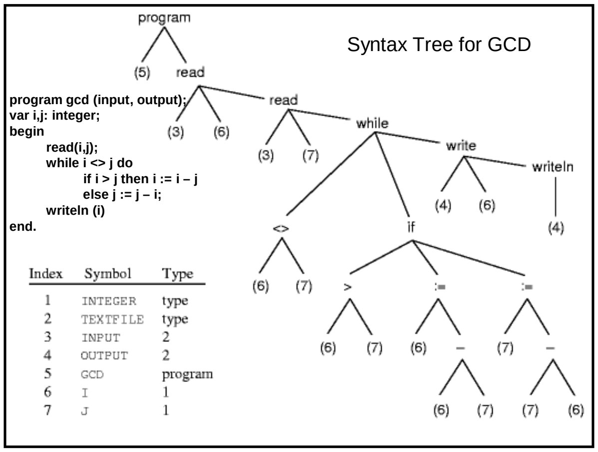
CJUMP ==
CX
CONST
0
MOVE
DX
CX
NOP

Code generation

CMP CX, 0
CMOVZ DX,CX



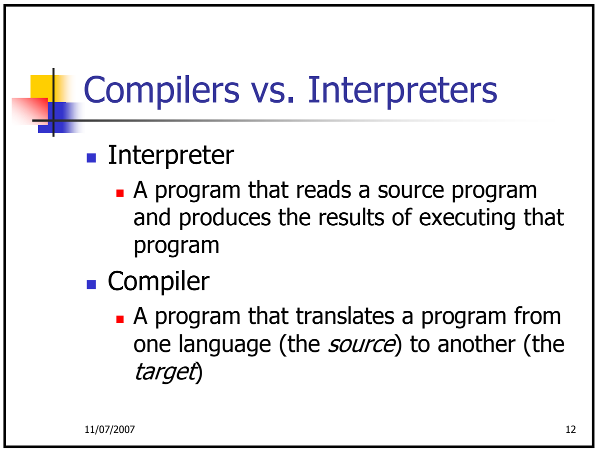
```
program gcd (input, output);
var i,j: integer;
begin
  read(i,j);
  while i <> j do
    if i > j then i := i – j
    else j := j – i;
  writeln (i)
end.
```



Syntax Tree for GCD

```
program gcd (input, output);
var i,j: integer;
begin
        read(i,j);
        while i <> j do
                if i > j then i := i – j
                else j := j – i;
        writeln (i)
end.
```

Index	Symbol	Type
1	INTEGER	type
2	TEXTFILE	type
3	INPUT	2
4	OUTPUT	2
5	GCD	program
6	I	1
7	J	1



Compilers vs. Interpreters

- Interpreter
  - A program that reads a source program and produces the results of executing that program
- Compiler
  - A program that translates a program from one language (the *source*) to another (the *target*)

Common Issues

- Compilers and interpreters both must read the input – a stream of characters – and “understand” it; *analysis*

```
w h i l e ( k < l e n g t h ) { <nl> <tab> i f ( a [ k ] > 0
) <nl> <tab> <tab>{ n P o s + + ; } <nl> <tab> }
```

Interpreter

- Interpreter
  - Execution engine
  - Program execution interleaved with analysis

```
running = true;
while (running) {
    analyze next statement;
    execute that statement;
}
```

  - May involve repeated analysis of some statements (loops, functions)

Compiler

- Read and analyze entire program
- Translate to semantically equivalent program in another language
  - Presumably easier to execute or more efficient
  - Should “improve” the program in some fashion
- Offline process
  - Tradeoff: compile time overhead (preprocessing step) vs execution performance

Typical Implementations

- Compilers
  - FORTRAN, C, C++, Java, COBOL, etc.
  - Strong need for optimization in many cases
- Interpreters
  - PERL, Python, Ruby, awk, sed, sh, csh, postscript printer, Scheme, Java VM
  - Effective if interpreter overhead is low relative to execution cost of individual statements

Hybrid approaches

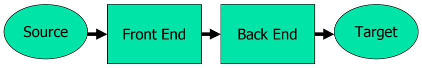
- Well-known example: Java
  - Compile Java source to byte codes – Java Virtual Machine language (.class files)
  - Execution
    - Interpret byte codes directly, or
    - Compile some or all byte codes to native code
      - Just-In-Time compiler (JIT) – detect hot spots & compile on the fly to native code
- Variation: .NET
  - Compilers generate MSIL
  - All IL compiled to native code before execution

Why Study Compilers?

- Better Understanding Of Implementation Issues in Programming Languages:
  - How Is “This” Implemented?
  - Why Does “This” Run So Slowly?
- Translation appears several places:
  - Processing command line parameters
  - Converting files/programs from one language/format to another

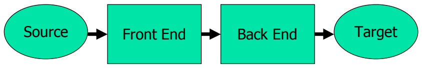
Structure of a Compiler

- First approximation
  - Front end: analysis
    - Read source program and understand its structure and meaning
  - Back end: synthesis
    - Generate equivalent target language program



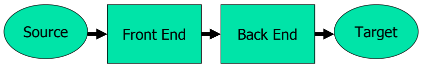
Implications

- Must recognize legal programs (& complain about illegal ones)
- Must generate correct code
- Must manage storage of all variables
- Must agree with OS & linker on target format

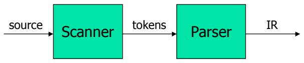


More Implications

- Need some sort of Intermediate Representation(s) (IR)
- Front end maps source into IR
- Back end maps IR to target machine code
- May be multiple IRs – higher level at first, lower level in later phases



Front End



- Split into two parts
  - Scanner: Responsible for converting character stream to token stream
    - Also strips out white space, comments
  - Parser: Reads token stream; generates IR
- Both of these can be generated automatically
  - Source language specified by a formal grammar
  - Tools read the grammar and generate scanner & parser (either table-driven or hard-coded)

Tokens

- Token stream: Each significant lexical chunk of the program is represented by a token
  - Operators & Punctuation: {}[]!+-=*;: …
  - Keywords: if while return goto
  - Identifiers: id & actual name
  - Constants: kind & value; int, floating-point character, string, …

Scanner Example

- Input text

```
// this statement does very little
if (x >= y) y = 42;
```

- Token Stream

IF LPAREN ID(x) GEQ ID(y)

RPAREN ID(y) BECOMES INT(42) SCOLON

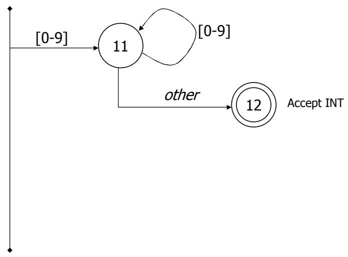
  - Notes: tokens are atomic items, not character strings; comments are *not* tokens

Example

- Possible syntax for numeric constants

digit ::= [0-9]
digits ::= *digit*+
number ::= *digits* (. *digits*)?
([eE] (+ | -)? digits) ?

Scanner DFA Example



Parser Output (IR)

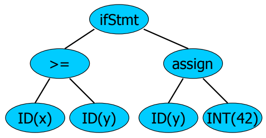
- Many different forms
  - Engineering tradeoffs that have changed over time
- Common output from a parser is an abstract syntax tree
  - Essential meaning of the program without the syntactic noise

Parser Example

- Token Stream Input

IF LPAREN ID(x)
GEQ ID(y) RPAREN
ID(y) BECOMES
INT(42) SCOLON

- Abstract Syntax Tree



Context-Free Grammars

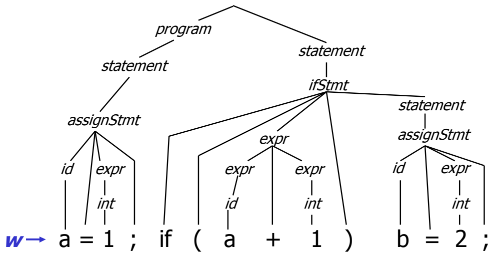
- Formally, a grammar *G* is a tuple *<N,Σ,P,S>* where
  - *N* a finite set of non-terminal symbols
  - Σ a finite set of terminal symbols
  - *P* a finite set of productions
    - A subset of $N \times (N \cup \Sigma)^*$
  - *S* the *start symbol,* a distinguished element of *N*
    - If not specified otherwise, this is usually assumed to be the non-terminal on the left of the first production

Grammar for a Tiny Language

- *program* ::= *statement* | *program statement*
- *statement* ::= *assignStmt* | *ifStmt*
- *assignStmt* ::= *id* = *expr* ;
- *ifStmt* ::= if (*expr*) *stmt*
- *expr* ::= *id* | *int* | *expr* + *expr*
- *Id* ::= a | b | c | i | j | k | n | x | y | z
- int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Example

G
program ::= *statement* | *program statement*
statement ::= *assignStmt* | *ifStmt*
assignStmt ::= *id* = *expr* ;
ifStmt ::= if (*expr*) *stmt*
expr ::= *id* | *int* | *expr* + *expr*
Id ::= a | b | c | i | j | k | n | x | y | z
int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9



Static Semantic Analysis

- During or (more common) after parsing
  - Type checking
  - Check for language requirements like proper declarations, type compatibility
  - Preliminary resource allocation
  - Collect other information needed by back end analysis and code generation

Back End

- Responsibilities
  - Translate IR into target machine code
  - Should produce fast, compact code
  - Should use machine resources effectively
    - Registers
    - Instructions
    - Memory hierarchy

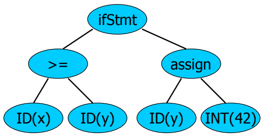
Back End Structure

- Typically split into two major parts with sub phases
  - "Optimization" – code improvements
    - May well translate parser IR into other IRs
    - We probably won't have time to do much with this part of the compiler, alas
  - Code generation
    - Instruction selection & scheduling
    - Register allocation

The Result

- Input

```
if (x >= y)
  y = 42;
```



- Output

```
  mov   eax,[ebp+16]
  cmp   eax,[ebp-8]
  jl     L17
  mov    [ebp-8],42
L17:
```