

# What's a Compiler???

## CSc 520

# Principles of Programming Languages

## 2: Translators

Christian Collberg

collberg+520@gmail.com

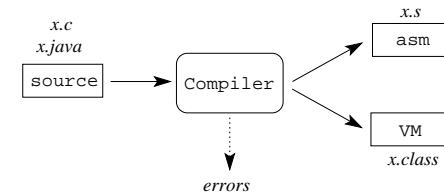
Department of Computer Science  
University of Arizona

Copyright © 2008 Christian Collberg

—Spring 2008 —2

[1]

- At the very basic level a compiler translates a computer program from source code to some kind of executable code:



- Often the source code is simply a text file and the executable code is a resulting assembly language program: `gcc -S x.c` reads the C source file `x.c` and generates an assembly code file `x.s`. Or the output can be a **virtual machine** code: `javac x.java` produces `x.class`.

520 —Spring 2008 —2

[2]

# What's a Language Translator???

- A compiler is really a special case of a **language translator**.
- A translator is a program that transforms a “program”  $P_1$  written in a language  $L_1$  into a program  $P_2$  written in another language  $L_2$ .
- Typically, we desire  $P_1$  and  $P_2$  to be semantically equivalent, i.e. they should behave identically.

# Example Language Translators

source language	translator	target language
$\text{\LaTeX}$	$\text{latex2html} \rightarrow$	html
Postscript	$\text{ps2ascii} \rightarrow$	text
FORTRAN	$\text{f2c} \rightarrow$	C
C++	$\text{cfront} \rightarrow$	C
C	$\text{gcc} \rightarrow$	assembly
.class	$\text{SourceAgain} \rightarrow$	Java
x86 binary	$\text{fx32} \rightarrow$	Alpha binary

—Spring 2008 —2

520 —Spring 2008 —2

# Compiler Input

**Text File** Common on Unix.

**Syntax Tree** A structure editor uses its knowledge of the source language syntax to help the user edit & run the program. It can send a syntax tree to the compiler, relieving it of lexing & parsing.

# Compiler Output

**Assembly Code** Unix compilers do this. Slow, but easy for the compiler.

**Object Code** .o-files on Unix. Faster, since we don't have to call the assembler.

**Executable Code** Called a **load-and-go**-compiler.

**Abstract Machine Code** Serves as input to an **interpreter**. Fast turnaround time.

**C-code** Good for portability.

# Compiler Tasks

**Static Semantic Analysis** Is the program (statically) correct? If not, produce error messages to the user.

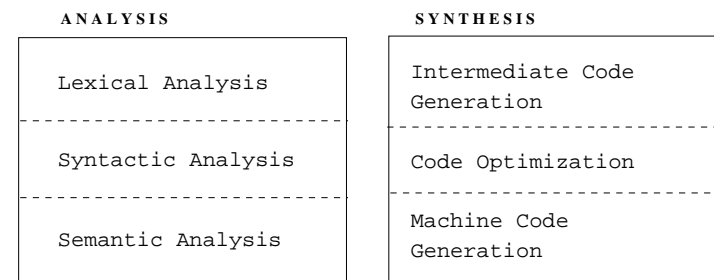
**Code Generation** The compiler must produce code that can be executed.

**Symbolic Debug Information** The compiler should produce a description of the source program needed by symbolic debuggers. Try `man gdb`.

**Cross References** The compiler may produce **cross-referencing** information. Where are identifiers declared & referenced?

**Profiler Information** Where does my program spend most of its execution time? Try `man gprof`.

# Compiler Phases



# Compiler Phases – Lexical analysis

- The lexer reads the source file and divides the text into lexical units (tokens), such as:

**Reserved words** BEGIN, IF,...

**identifiers** x, StringTokenizer,...

**special characters** +, \*, -, ^, ...

**numbers** 30, 3.14, ...

**comments** (\* text \*),

**strings** "text".

- Lexical errors (such as 'illegal character', 'undelimited character string', 'comment without end') are reported.

# Lexical analysis — Example

## Lexical Analysis of English

- The sentence

The boy's cowbell won't play.

would be translated to the list of tokens

the, boy+possessive, cowbell, will, not, play

## Lexical Analysis of Java

- The sentence

x = 3.14 \* (9.0+y);

would be translated to the list of tokens

<ID,x>, EQ, <FLOAT,3.14>, STAR, LPAREN,  
<FLOAT,9.0>, PLUS, <ID,y>, RPAREN, SEMICOLON

# Compiler Phases – Syntactic analysis

- Syntax analysis (**parsing**) determines the **structure** of a sentence.

- The compiler reads the tokens produced during lexical analysis and builds an **abstract syntax tree** (AST), which reflects the hierarchical structure of the input program.

- Syntactic errors are reported to the user:

- 'missing ;',
- 'BEGIN without END'

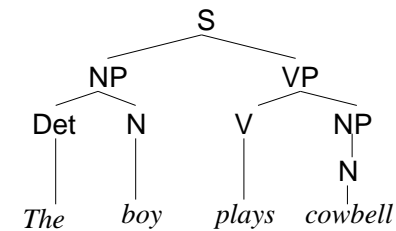
# Syntactic analysis — Example

## Syntactic Analysis of English

- The sentence

The boy plays cowbell.

would be parsed into the tree



- S=sentence, NP=noun phrase, VP=verb phrase, Det=determiner, N=noun, V=verb.

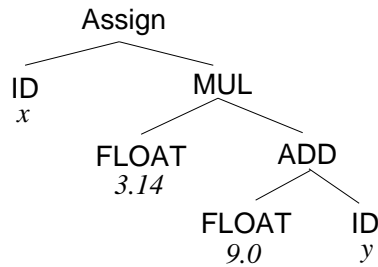
# Syntactic analysis — Example

## Syntactic Analysis of Java

- The sentence

```
x = 3.14 * (9.0+y);
```

would be parsed into the AST



# Compiler Phases – Semantic analysis

- The AST produced during syntactic analysis is decorated with **attributes**, which are then evaluated. The attributes can represent any kind of information such as expression types.
- The compiler also collects information useful for future phases.
- Semantic errors are reported:
  - 'identifier not declared',
  - 'integer type expected'.

# Semantic analysis — Example

## Semantic Analysis of English

- In the sentence

```
The boy plays his cowbell.
```

we determine that **his** refers to **the boy**.

## Semantic Analysis of Java

- In the sentence

```
static float luftballons = 10;
void P() {int luftballons = 99;
         System.out.println(luftballons)}
```

the compiler must determine

- which **luftballons** the print-statement refers to,
- that **float luftballons=10** has the wrong type.

# Compiler Phases – IR Generation

- From the decorated AST this phase generates **intermediate code** (IR).
- The IR adds an extra level of abstraction between the high level AST and the very low level assembly code we want to generate. This simplifies code generation.
- A carefully designed IR allows us to build compilers for a number of languages and machines, by mixing and matching front-ends and back-ends.

# IR Generation — Example

## IR Generation of English

- From the sentence

Every boy has a cowbell.

we could generate

$$\forall x; \text{boy}(x) \Rightarrow \text{has-cowbell}(x)$$

## IR Generation of Java

- From the sentence

$x = 3.14 * (9.0 + y);$

the compiler could generate the (stack-based) IR code

```
pusha x, pushf 3.14, pushf 9.0,  
push y, add, mul, assign
```

# Compiler Phases – Code Optimization

- The (often optional) code optimization phase transforms an IR program into an equivalent but more efficient program.
- Typical transformations include
  - common subexpression elimination** only compute an expression once, even if it occurs more than once in the source),
  - inline expansion** insert a procedure's code at the call site to reduce call overhead,
  - algebraic transformations**  $A := A + A$  is faster than  $A := A * 2$ .

# Compiler Phases – Code Generation

- The last compilation phase transforms the intermediate code into machine code, usually assembly code or link modules.
- Alternatively, the compiler generates **Virtual Machine Code** (VM), i.e. code for a software defined architecture. Java compilers, for example, generate class files containing bytecodes for the Java VM.

# Multi-pass Compilation

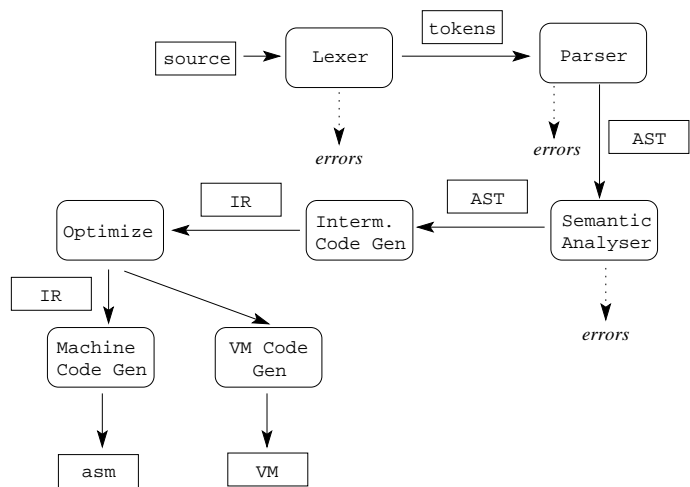
- The next slide shows the outline of a typical compiler. In a unix environment each pass could be a stand-alone program, and the passes could be connected by pipes:

```
lex x.c | parse | sem | ir | opt | codegen > x.s
```
- For performance reasons the passes are usually integrated:

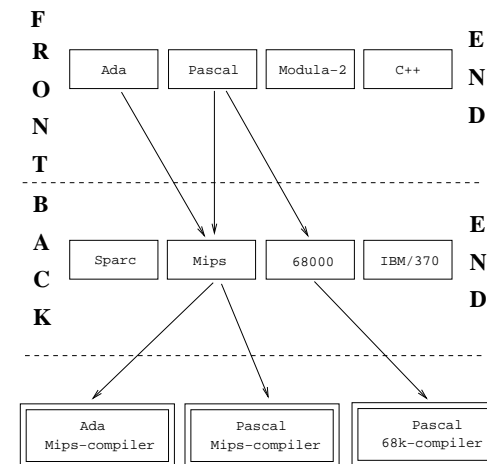
```
front x.c > x.ir  
back x.ir > x.s
```

The front-end does all analysis and IR generation. The back-end optimizes and generates code.

# Multi-pass Compilation...



# Mix-and-Match Compilers



## Example

- Let's compile the procedure Foo, from start to finish:

```

PROCEDURE Foo ();
VAR i : INTEGER;
BEGIN
  i := 1;
  WHILE i < 20 DO
    PRINT i * 2;
    i := i * 2 + 1;
  ENDDO;
END Foo;

```

The compilation phases are:

Lexial Analysis ⇒ Syntactic Analysis ⇒ Semantic Analysis ⇒ Intermediate code generation ⇒ Code Optimization ⇒ Machine code generation.

## Example – Lexical Analysis

- Break up the source code (a text file) and into tokens.

Source Code	Stream of Tokens
PROCEDURE Foo ();	PROCEDURE, <id,Foo>, LPAR, RPAR, SC,
VAR i : INTEGER;	VAR, <id,i>, COLON, <id,INTEGER>,SC,
BEGIN	BEGIN, <id,i>,CEQ,<int,1>,SC,
i := 1;	WHILE, <id,i>, LT, <int,20>,DO,
WHILE i < 20 DO	PRINT, <id,i>, MUL, <int,2>, SC,
PRINT i * 2;	<id,i>, CEQ, <id,i>, MUL, <int,2>, PLUS,
i := i * 2 + 1;	<int,1>, SC, ENDDO, SC, END, <id,Foo>, SC
ENDDO;	
END Foo;	

# Example – Lexical Analysis...

- We defined the following set of tokens:

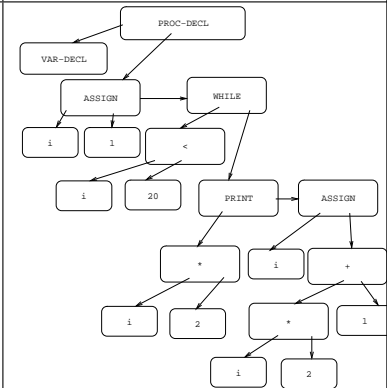
TOKEN	STRING	TOKEN	STRING
PROCEDURE	"PROCEDURE"	<int,1>	<b>integer literal</b>
<id, Foo>	<b>identifier</b>	WHILE	"WHILE"
LPAR	"("	LT	"<"
RPAR	")"	DO	"DO"
SC	";"	PRINT	"PRINT"
VAR	"VAR"	MUL	"*"
COLON	":"	PLUS	"+"
BEGIN	"BEGIN"	ENDDO	"ENDDO"
CEQ	":="	END	"END"

# Example – Syntactic Analysis

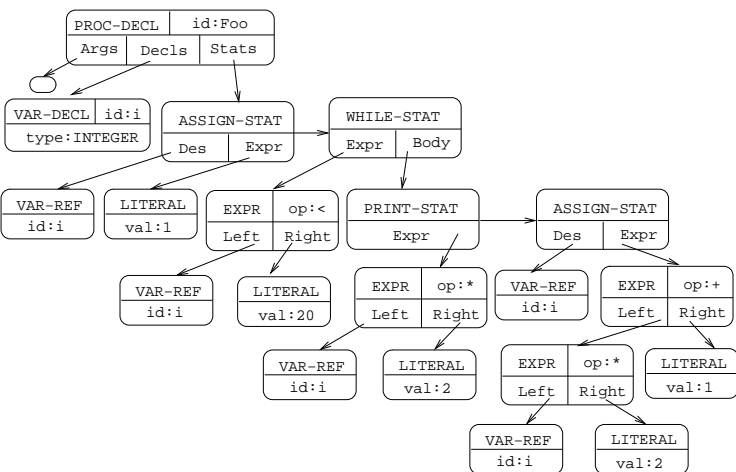
## Stream of Tokens

PROCEDURE, <id, Foo>, LPAR, RPAR, SC, VAR, <id, i>, COLON, <id, INTEGER>, SC, BEGIN, <id, i>, CEQ, <int, 1>, SC, WHILE, <id, i>, LT, <int, 20>, DO, PRINT, <id, i>, MUL, <int, 2>, SC, <id, i>, CEQ, <id, i>, MUL, <int, 2>, PLUS, <int, 1>, SC, ENDDO, SC, END, <id, Foo>, SC

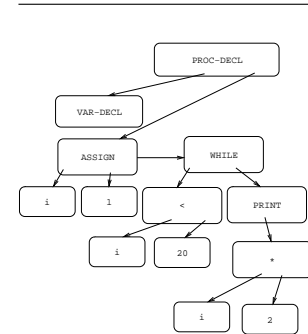
## Abstract Syntax Tree



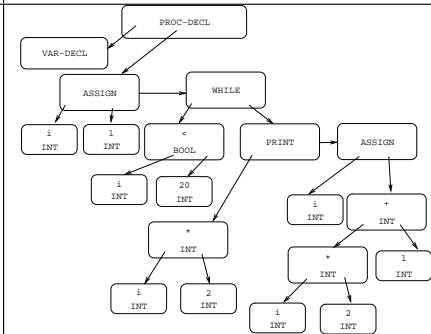
# Example – Semantic Analysis



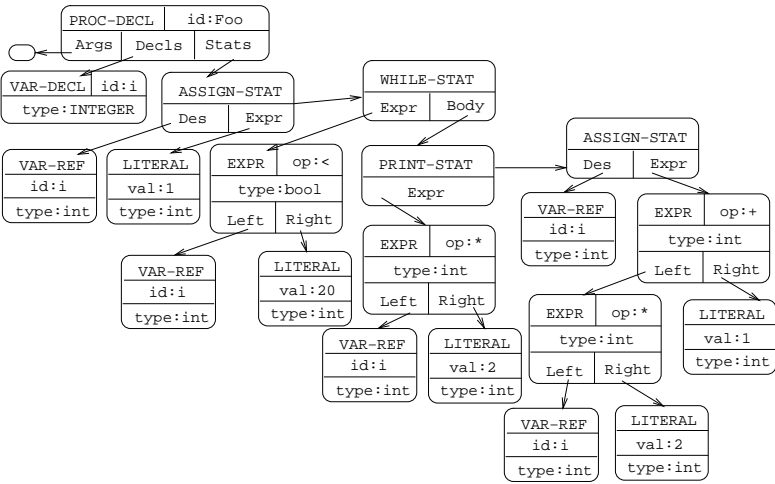
## Abstract Syntax Tree



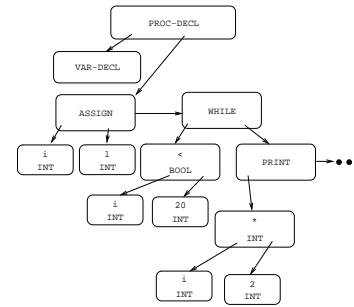
## Decorated Abstract Syntax Tree



# Example – IR Generation



Decorated Abstract Syntax Tree



Intermediate Code

[1]	ASSIGN	i	1
[2]	BRGE	i	20 [9]
[3]	MUL	$t_1$	i 2
[4]	PRINT	$t_1$	
[5]	MUL	$t_2$	i 2
[6]	ADD	$t_3$	$t_2$ 1
[7]	ASSIGN	i	$t_3$
[8]	JUMP	[2]	
[9]			

# Example – IR Generation...

Intermediate Code	Intermediate Code Definition
[1] ASSIGN i 1	<b>ASSIGN</b> $A, B$ $A := B$ ;
[2] BRGE i 20 [9]	<b>BRGE</b> $A, B, C$ IF ( $A \geq B$ ) THEN continue at instruction $C$ ;
[3] MUL $t_1$ i 2	<b>MUL</b> $A, B, C$ $A := B * C$ ;
[4] PRINT $t_1$	<b>ADD</b> $A, B, C$ $A := B + C$ ;
[5] MUL $t_2$ i 2	<b>SHL</b> $A, B, C$ $A := \text{shift } B \text{ left } C$ steps;
[6] ADD $t_3$ $t_2$ 1	<b>PRINT</b> $A$ Print $A$ and a newline;
[7] ASSIGN i $t_3$	<b>JUMP</b> $A$ Continue at instruction $A$ ;
[8] JUMP [2]	
[9]	

# Example – Code Optimization

Intermediate Code	Optimized Intermediate Code
[1] ASSIGN i 1	[1] ASSIGN i 1
[2] BRGE i 20 [9]	[2] BRGE i 20 [8]
[3] MUL $t_1$ i 2	[3] SHL $t_1$ i 1
[4] PRINT $t_1$	[4] PRINT $t_1$
[5] MUL $t_2$ i 2	[5] ADD $t_2$ $t_1$ 1
[6] ADD $t_3$ $t_2$ 1	[6] ASSIGN i $t_2$
[7] ASSIGN i $t_3$	[7] JUMP [2]
[8] JUMP [2]	[8]
[9]	

# Example – Machine Code Generation

Intermediate Code	MIPS Machine Code
[1] ASSIGN i 1	.data _i: .word 0
[2] BRGE i 20 [8]	.text
[3] SHL t <sub>1</sub> i 1	.globl main
[4] PRINT t <sub>1</sub>	main: li \$14, 1
[5] ADD t <sub>2</sub> t <sub>1</sub> 1	\$32: bge \$14, 20, \$33
[6] ASSIGN i t <sub>2</sub>	sll \$a0, \$14, 1
[7] JUMP [2]	li \$v0, 1
[8]	syscall
	addu \$14, \$a0, 1
	b \$32
	\$33: sw \$14, _i

# Interpretation

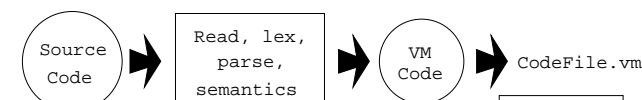
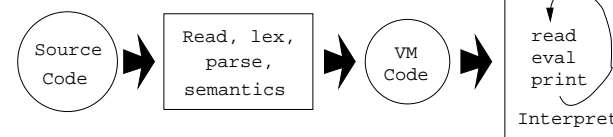
- An interpreter is like a CPU, only in software.
- The compiler generates *virtual machine* (VM) code rather than native machine code.
- The interpreter executes VM instructions rather than native machine code.

## Interpretation...

- Interpreters are **slow** Often 10–100 times slower than executing machine code directly.
- Interpreters are **portable** The virtual machine code is not tied to any particular architecture.
- Interpreters work well with very high-level, dynamic languages (APL, Prolog, ICON) where a lot is unknown at compile-time (array bounds, etc).

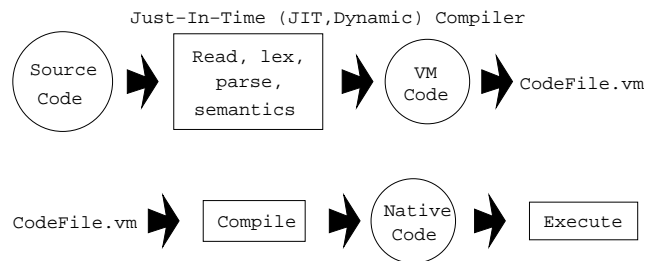
## Kinds of Interpreters

"APL/Prolog-style (load-and-go/interactive) interpreter"



"Java-style interpreter"

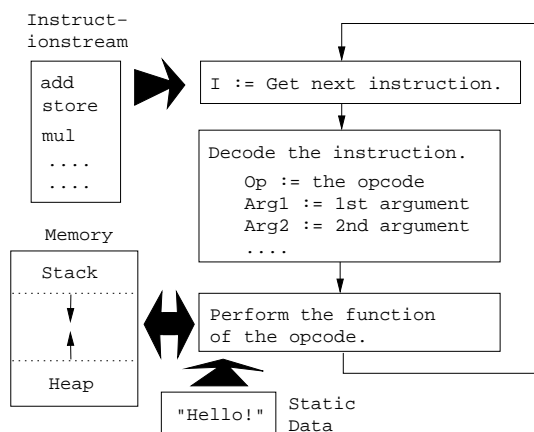
# Kinds of Interpreters...



# Actions in an Interpreter

- Internally, an interpreter consists of
  - The interpreter *engine*, which executes the VM instructions.
  - Memory* for storing user data. Often separated as a heap and a stack.
  - A stream of VM instructions.

# Actions in an Interpreter...



# Readings and References

- Read: Michael Scott, *Programming Language Pragmatics*, pages 15–29.

## Summary

- The structure of a compiler depends on
  1. the complexity of the language we're working on (higher complexity  $\Rightarrow$  more passes),
  2. the quality of the code we hope to produce (better code  $\Rightarrow$  more passes),
  3. the degree of portability we hope to achieve (more portable  $\Rightarrow$  better separation between front- and back-ends).
  4. the number of people working on the compiler (more people  $\Rightarrow$  more independent modules).

## Summary...

- Some highly retargetable compilers for high-level languages produce C-code, rather than machine code. This C-code is then compiled by the native C compiler to machine code.
- Some languages (APL, LISP, Smalltalk, Java, ICON, Perl, Awk) are traditionally **interpreted** (executed in software by an **interpreter**) rather than compiled to machine code.

## Summary...

- Some interpreters use **dynamic compilation** (or **jitting**), switching between
  1. interpreting the virtual machine code,
  2. translating the virtual machine code to native machine code,
  3. executing the native machine code,
  4. optimizing the native and/or virtual machine code, and
  5. throwing native code away if it is no longer needed or takes up too much room.All this is done dynamically at runtime.