Programming Language Translators

Jordi Cortadella

Credits

Most of the material in these slides has been extracted from the one elaborated by Prof. Stephen A. Edwards (University of Columbia) for the course COMS W4115 (Programming Languages and Translators)

What's in a Language?

Components of a language: Syntax

How characters combine to form words, sentences, paragraphs.

The quick brown fox jumps over the lazy dog.

is syntactically correct English, but isn't a Java program.

```
class Foo {
  public int j;
  public int foo(int k) { return j + k; }
}
```

Is syntactically correct Java, but isn't C.

Specifying Syntax

Usually done with a context-free grammar.

Typical syntax for algebraic expressions:

$$
\begin{aligned}
expr \quad \rightarrow \quad & expr + expr \\
| \quad & expr - expr \\
| \quad & expr * expr \\
| \quad & expr / expr \\
| \quad & \textbf{digit} \\
| \quad & (expr)
\end{aligned}
$$

Components of a language: Semantics

What a well-formed program "means."

The semantics of C says this computes the nth Fibonacci number.

```c
int fib(int n)
{
  int a = 0, b = 1;
  int i;
  for (i = 1 ; i < n ; i++) {
    int c = a + b;
    a = b; b = c;
  }
  return b;
}
```

Semantics

Something may be syntactically correct but semantically nonsensical.

The rock jumped through the hairy planet.

Or ambiguous

The chickens are ready for eating.

Semantics

Nonsensical in Java:

```
class Foo {
  int bar(int x) { return Foo; }
}
```

Ambiguous in Java:

```
class Bar {
  public float foo() { return 0; }
  public int foo() { return 0; }
}
```

Specifying Semantics

Doing it formally beyond the scope of this class, but basically two ways:

- Operational semantics: Define a virtual machine and how executing the program evolves the state of the virtual machine

- Denotational semantics: Shows how to build the function representing the behavior of the program (i.e., a transformation of inputs to outputs) from statements in the language.

Most language definitions use an informal operational semantics written in English.

Language Processors

Interpreter

Compiler

```
           ┌──────────────────┐
           │  Source Program  │
           └──────────────────┘
                    │
                    ▼
             ┌──────────────┐
             │   Compiler   │
             └──────────────┘
                    │
                    ▼
┌─────────┐   ┌───────────────────────┐   ┌──────────┐
│  Input  │──▶│  Executable Program   │──▶│  Output  │
└─────────┘   └───────────────────────┘   └──────────┘
```

Bytecode Interpreter

```
        ┌─────────────────────┐
        │   Source Program    │
        └─────────────────────┘
                   │
                   ▼
        ┌─────────────────────┐
        │      Compiler       │
        └─────────────────────┘
                   │
                   ▼
        ┌─────────────────────┐
        │      Bytecode       │
        └─────────────────────┘
                   │
                   ▼
┌────────┐  ┌───────────────────────┐  ┌────────┐
│ Input  │─▶│ Bytecode Interpreter  │─▶│ Output │
└────────┘  └───────────────────────┘  └────────┘
```

Just-in-time Compiler

```
                    ┌──────────────────┐
                    │  Source Program  │
                    └────────┬─────────┘
                             │
                        ┌────▼────┐
                        │ Compiler│
                        └────┬────┘
                             │
                       ┌─────▼─────┐
                       │ Bytecode  │
                       └─────┬─────┘
                             │
          ┌──────────────────▼────────────────────────────┐
          │         ┌──────────────────────────────┐      │
 ┌───────┐│         │   Just-in-time Compiler      │      │  ┌────────┐
 │ Input ├┼────────▶└──────────────┬───────────────┘      ├─▶│ Output │
 └───────┘│                   ┌────▼────────┐             │  └────────┘
          │                   │Machine Code │             │
          │                   └─────────────┘             │
          └───────────────────────────────────────────────┘
```
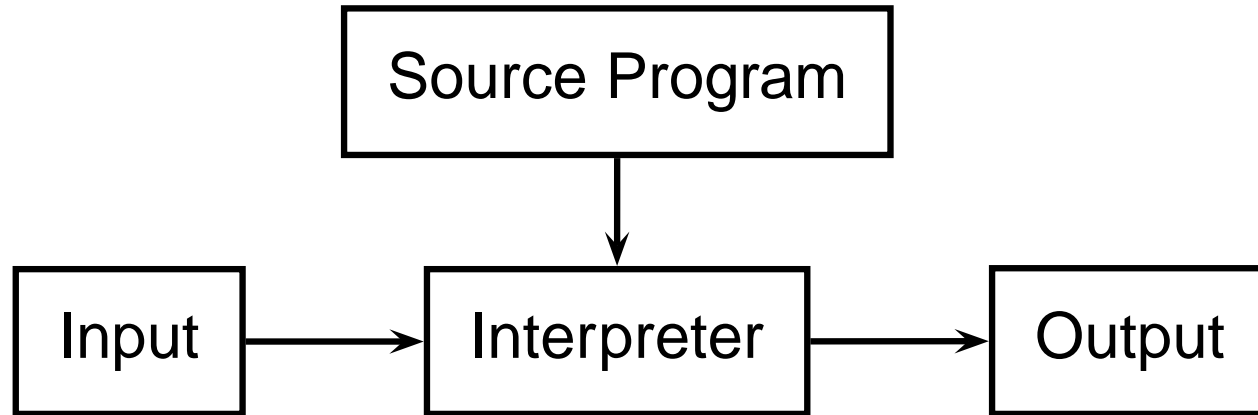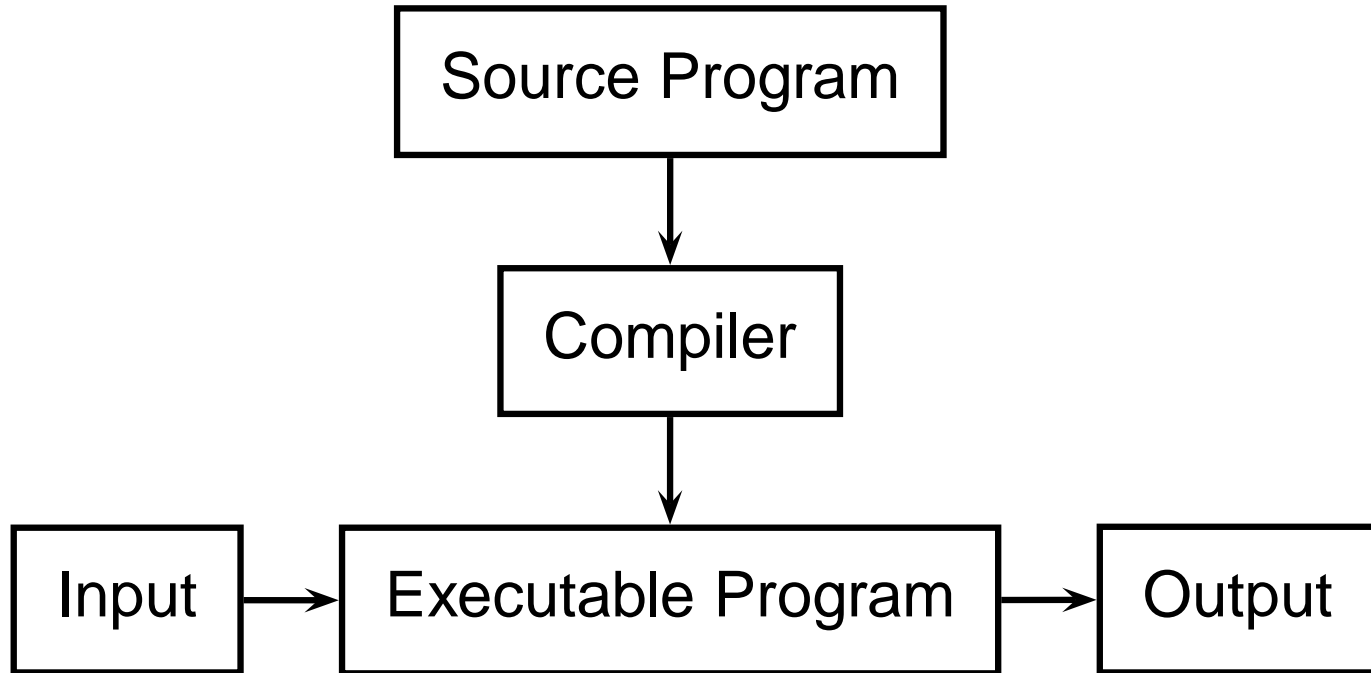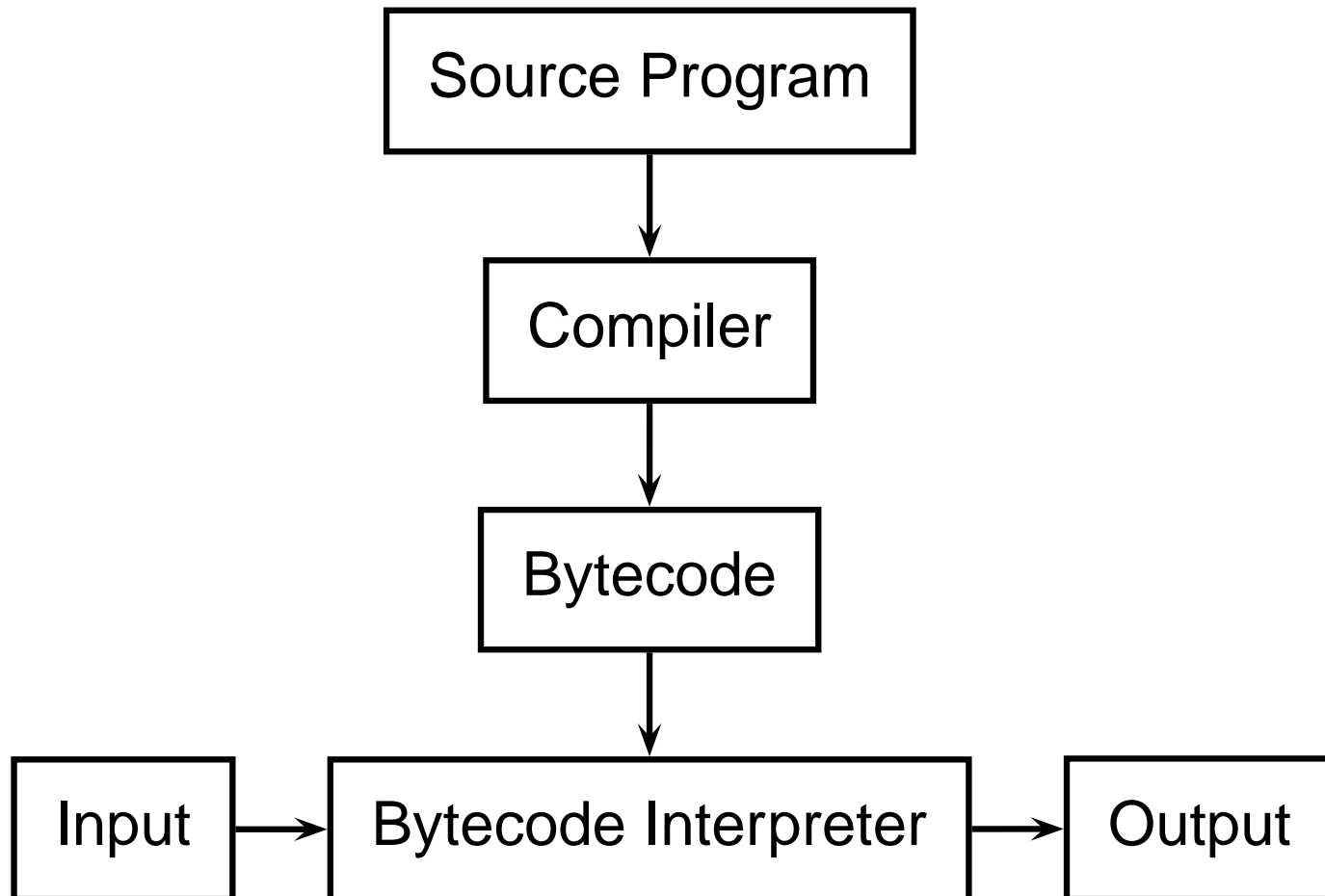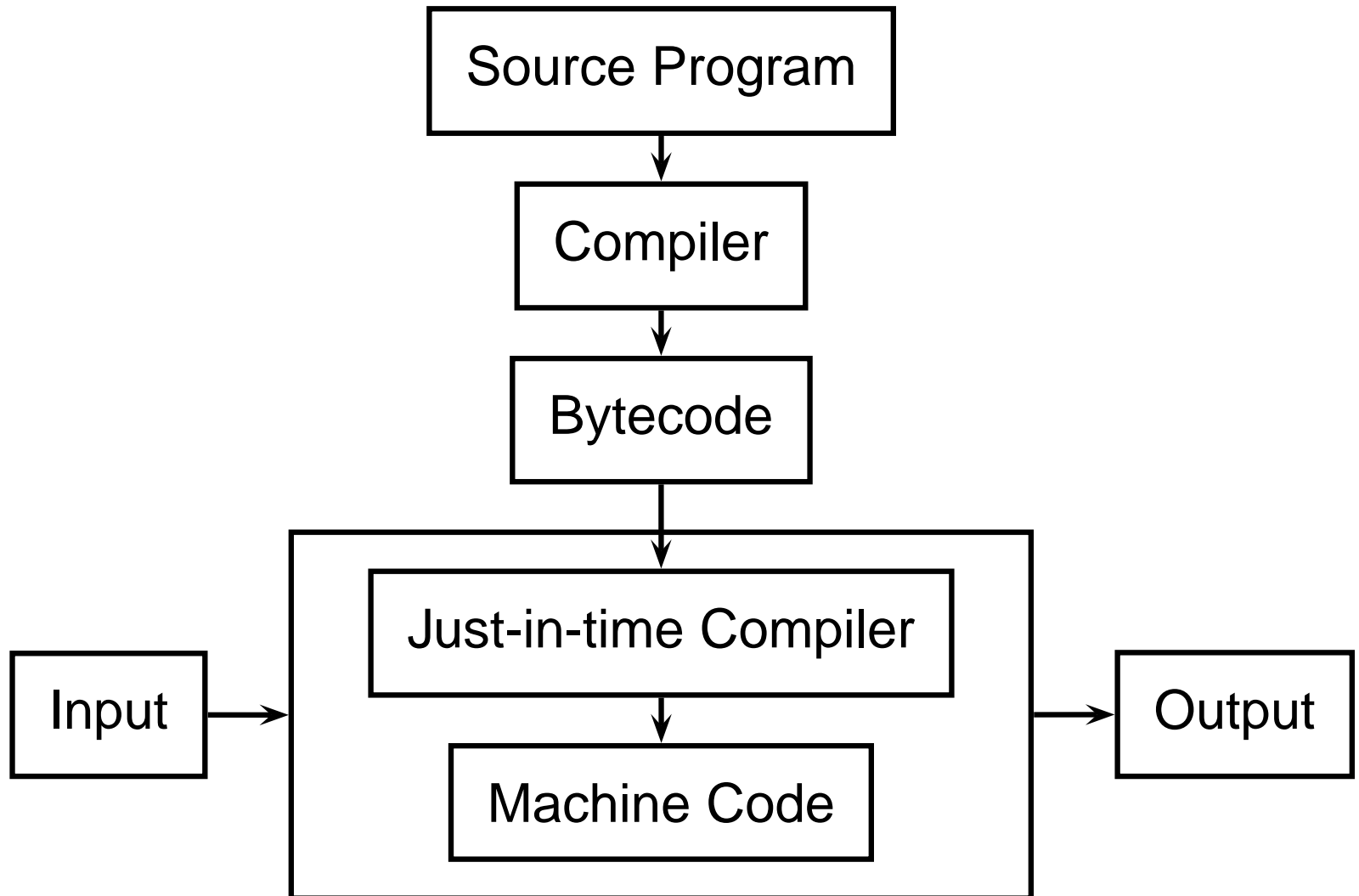
Language Speeds Compared

Language	Impl.
C	**gcc**
Ocaml	**ocaml**
SML	**mlton**
C++	**g++**
SML	**smlnj**
Common Lisp	**cmucl**
Scheme	**bigloo**
Ocaml	ocamlb
Java	**java**
Pike	pike
Forth	*gforth*
Lua	lua
Python	python
Perl	perl
Ruby	ruby
Eiffel	**se**
Mercury	**mercury**
Awk	mawk
Haskell	**ghc**
Lisp	rep
Icon	icon
Tcl	tcl
Javascript	njs
Scheme	guile
Forth	**bigforth**
Erlang	erlang
Awk	gawk
Emacs Lisp	xemacs
Scheme	**stalin**
PHP	php
Bash	bash

bytecodes   **native code**   **JIT**   *Threaded code*

http://www.bagley.org/~doug/shootout/

Separate Compilation

foo.c   bar.c

C compiler cc:

foo.s   bar.s   printf.o   fopen.o   malloc.o   · · ·

Assembler as:

Archiver ar:

foo.o   bar.o   · · ·   libc.a

Linker ld:

foo — An Executable

Preprocessor

"Massages" the input before the compiler sees it.

- Macro expansion

- File inclusion

- Conditional compilation

The C Preprocessor

```
#include <stdio.h>
#define min(x, y) \
  ((x)<(y))?(x):(y)
#ifdef DEFINE_BAZ
int baz();
#endif
void foo()
{
  int a = 1;
  int b = 2;
  int c;
  c = min(a,b);
}
```

cc -E example.c gives

```
extern int
printf(char*,...);
```
... many more declarations
from stdio.h

```
void foo()
{
  int a = 1;
  int b = 2;
  int c;
  c = ((a)<(b))?(a):(b);
}
```

Compiling a Simple Program

```c
int gcd(int a, int b)
{
  while (a != b) {
    if (a > b) a -= b;
    else b -= a;
  }
  return a;
}
```

What the Compiler Sees

```
int gcd(int a, int b)
{
  while (a != b) {
    if (a > b) a -= b;
    else b -= a;
  }
  return a;
}
```

i n t sp g c d (i n t sp a , sp i

n t sp b) nl { nl sp sp w h i l e sp

(a sp ! = sp b) sp { nl sp sp sp sp i

f sp (a sp > sp b) sp a sp - = sp b

; nl sp sp sp sp e l s e sp b sp - = sp

a ; nl sp sp } nl sp sp r e t u r n sp

a ; nl } nl

Text file is a sequence of characters
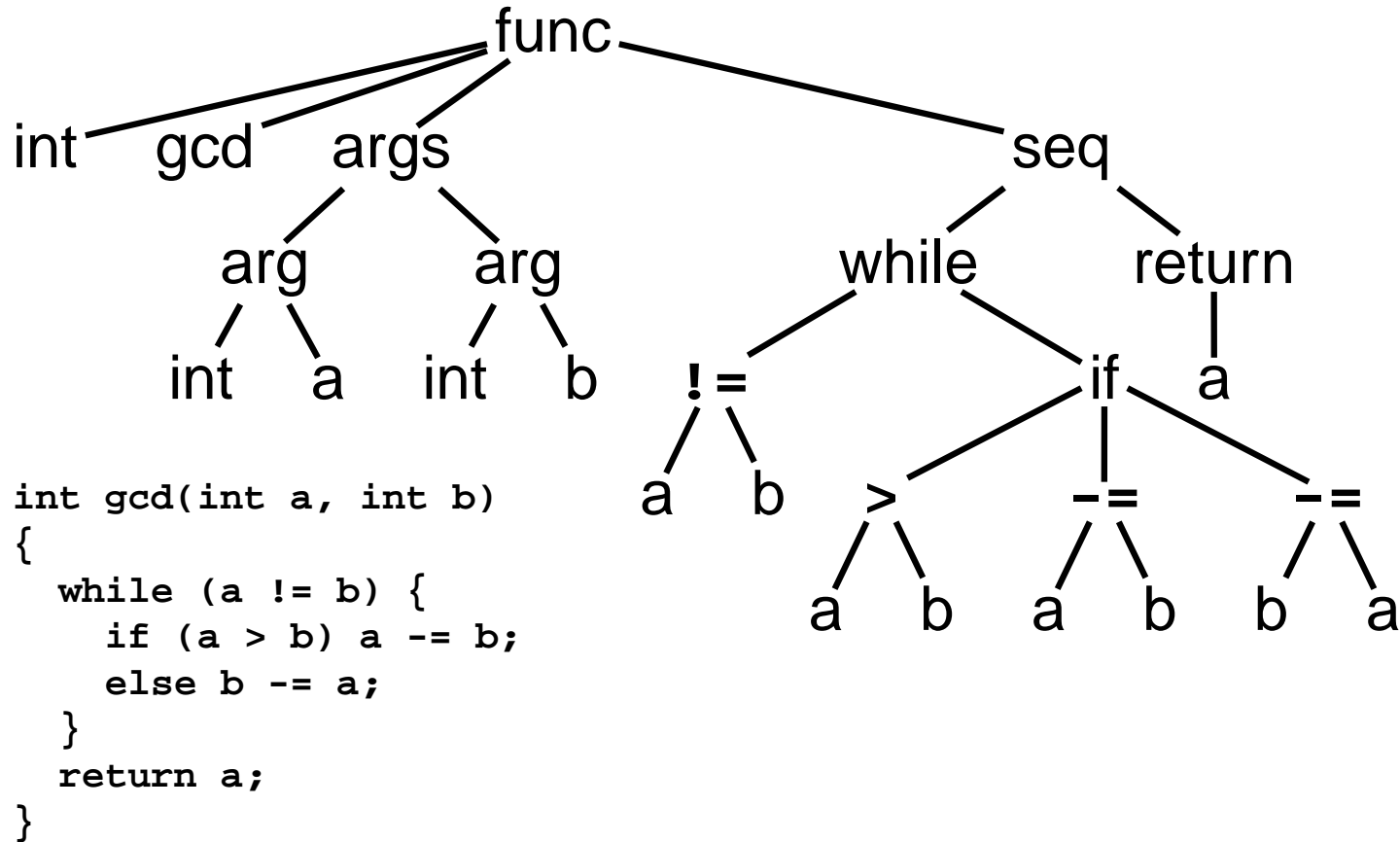
Lexical Analysis Gives Tokens

```
int gcd(int a, int b)
{
  while (a != b) {
    if (a > b) a -= b;
    else b -= a;
  }
  return a;
}
```
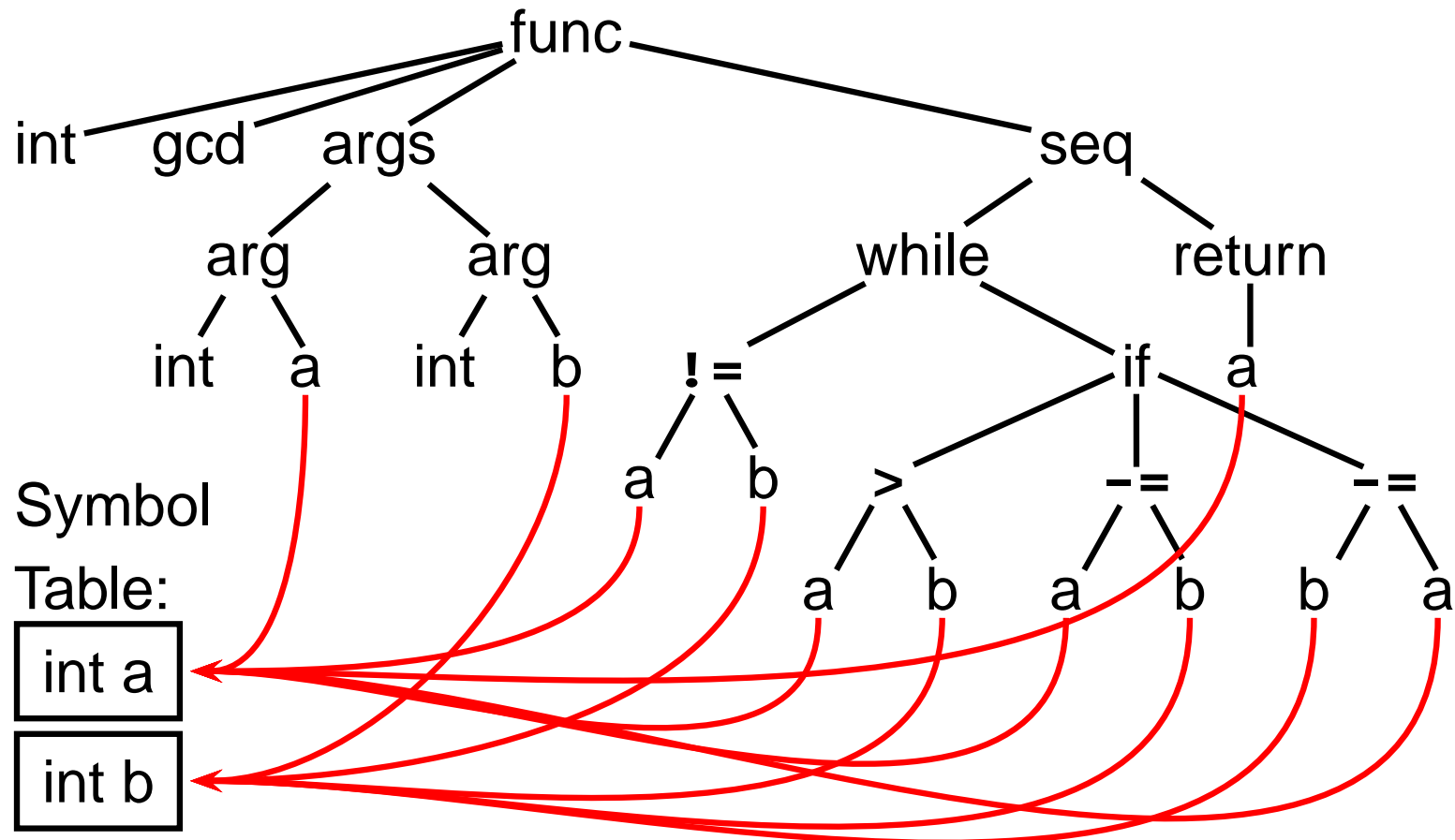
int	gcd	(	int	a	,	int	b	)	{

while	(	a	!=	b	)	{	if	(	a	>

b	)	a	-=	b	;	else	b	-=	a	;

}	return	a	;	}
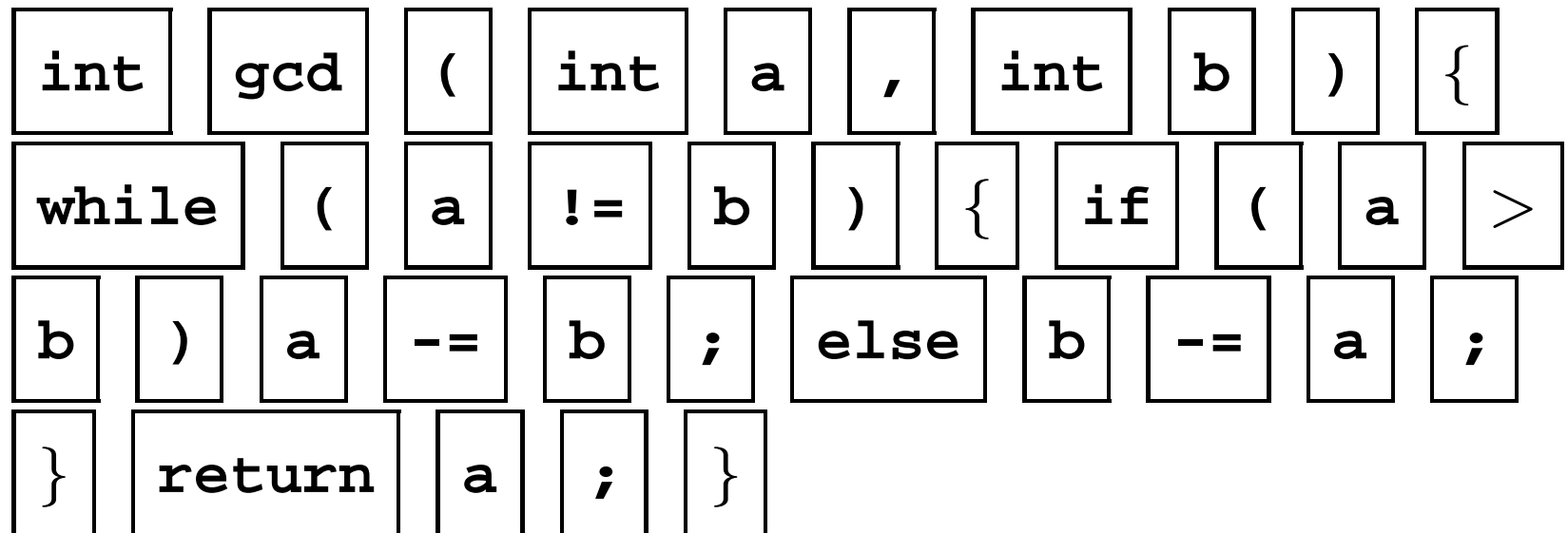
A stream of tokens. Whitespace, comments removed.

Parsing Gives an AST


```
int gcd(int a, int b)
{
  while (a != b) {
    if (a > b) a -= b;
    else b -= a;
  }
  return a;
}
```

Abstract syntax tree built from parsing rules.

Semantic Analysis Resolves Symbols

Types checked; references to symbols resolved

Translation into 3-Address Code

```
L0: sne    $1,  a, b
    seq    $0, $1, 0
    btrue  $0, L1     % while (a != b)
    sl     $3,  b, a
    seq    $2, $3, 0
    btrue  $2, L4     % if (a < b)
    sub    a,   a, b  % a -= b
    jmp    L5
L4: sub    b,   b, a  % b -= a
L5: jmp    L0
L1: ret    a
```

```
int gcd(int a, int b)
{
  while (a != b) {
    if (a > b) a -= b;
    else b -= a;
  }
  return a;
}
```

Idealized assembly language w/ infinite registers

Generation of 80386 Assembly

```
gcd:    pushl  %ebp                % Save FP
        movl   %esp,%ebp
        movl   8(%ebp),%eax        % Load a from stack
        movl   12(%ebp),%edx       % Load b from stack
.L8:    cmpl   %edx,%eax
        je     .L3                 % while (a != b)
        jle    .L5                 % if (a < b)
        subl   %edx,%eax           % a -= b
        jmp    .L8
.L5:    subl   %eax,%edx           % b -= a
        jmp    .L8
.L3:    leave                      % Restore SP, BP
        ret
```

Generation of Wasm

```
(module
 ….
 (func $myfunc (result i32)
   ….
   block
   loop
   ….
   get_local $x
   i32.const 50
   i32.eq
   br_if 1
   Br 0
 )
   ….
)
```

Compiler overview

position = initial + rate * 60

```
┌─────────────────────────────┐
│       Lexical Analyzer      │
└─────────────────────────────┘
```

$\langle \mathbf{id}, 1 \rangle \ \langle = \rangle \ \langle \mathbf{id}, 2 \rangle \ \langle + \rangle \ \langle \mathbf{id}, 3 \rangle \ \langle * \rangle \ \langle 60 \rangle$

```
┌─────────────────────────────┐
│       Syntax Analyzer       │
└─────────────────────────────┘
```

```
              =
      ⟨id,1⟩     +
            ⟨id,2⟩   *
                  ⟨id,3⟩   60
```

```
┌─────────────────────────────┐
│      Semantic Analyzer      │
└─────────────────────────────┘
```

```
              =
      ⟨id,1⟩     +
            ⟨id,2⟩   *
                  ⟨id,3⟩   inttofloat
                              |
                              60
```

```
┌─────────────────────────────────┐
│   Intermediate Code Generator   │
└─────────────────────────────────┘
```

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

```
┌─────────────────────────────┐
│       Code Optimizer        │
└─────────────────────────────┘
```

```
t1 = id3 * 60.0
id1 = id2 + t1
```

```
┌─────────────────────────────┐
│       Code Generator        │
└─────────────────────────────┘
```

```
LDF   R2, id3
MULF  R2, R2, #60.0
LDF   R1, id2
ADDF  R1, R1, R2
STF   id1, R1
```

1	position	⋯
2	initial	⋯
3	rate	⋯

SYMBOL TABLE