IN4: Stages of a Compiler

CMPT 379
Compilers

Anoop Sarkar

http://anoopsarkar.github.io/compilers-class/

IN4: Stages of a Compiler

Introduction to Compilers

CMPT 379: Compilers

Instructor: Anoop Sarkar

anoopsarkar.github.io/compilers-class

Building a compiler

- Programming languages have a lot in common
- Do not write a compiler for each language
- Create a general mathematical model for the **structure** of all languages
- Implement a compiler using this model

Building a compiler

- Each language compiler is built using this general model (so-called compiler compilers)
  - yacc = yet another compiler compiler
- Code optimization ideas can also be shared across languages

Demo: compiler for the expr language

Building a compiler

- The cost of compiling and executing should be managed
- No program that violates the definition of the language should escape
- No program that is valid should be rejected

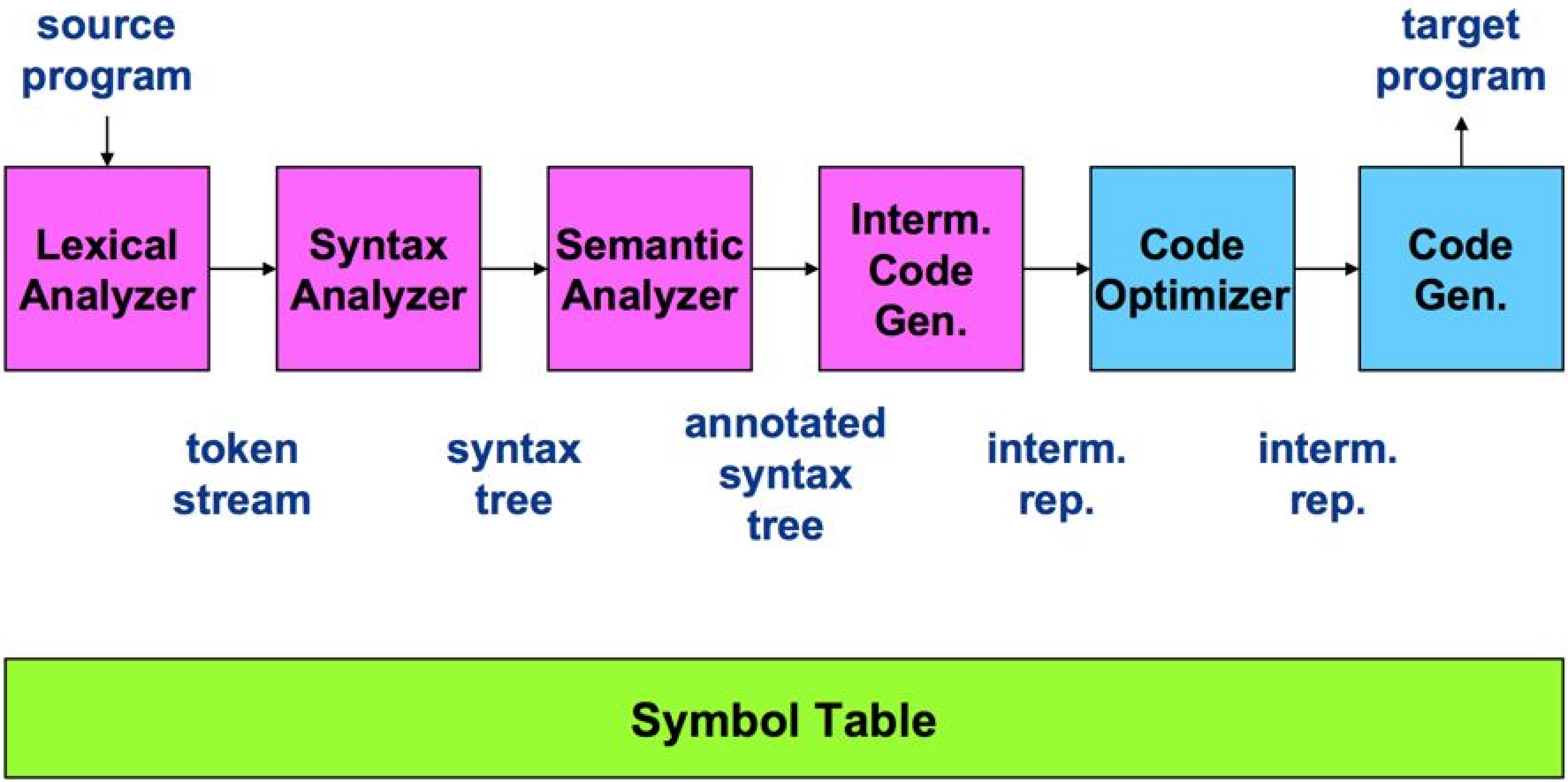
Building a compiler

- Requirements for building a compiler:
  - Symbol-table management
  - Error detection and reporting
- Stages of a compiler:
  - Analysis (front-end)
  - Synthesis (back-end)

Stages of a Compiler

- Analysis (Front-end)
  - Lexical analysis
  - Syntax analysis (parsing)
  - Semantic analysis (type-checking)
- Synthesis (Back-end)
  - Intermediate code generation
  - Code optimization
  - Code generation

Stages of a Compiler



Compiler Front-end

Lexical Analysis

- Also called *scanning*, take input program *string* and convert into tokens
- Example:

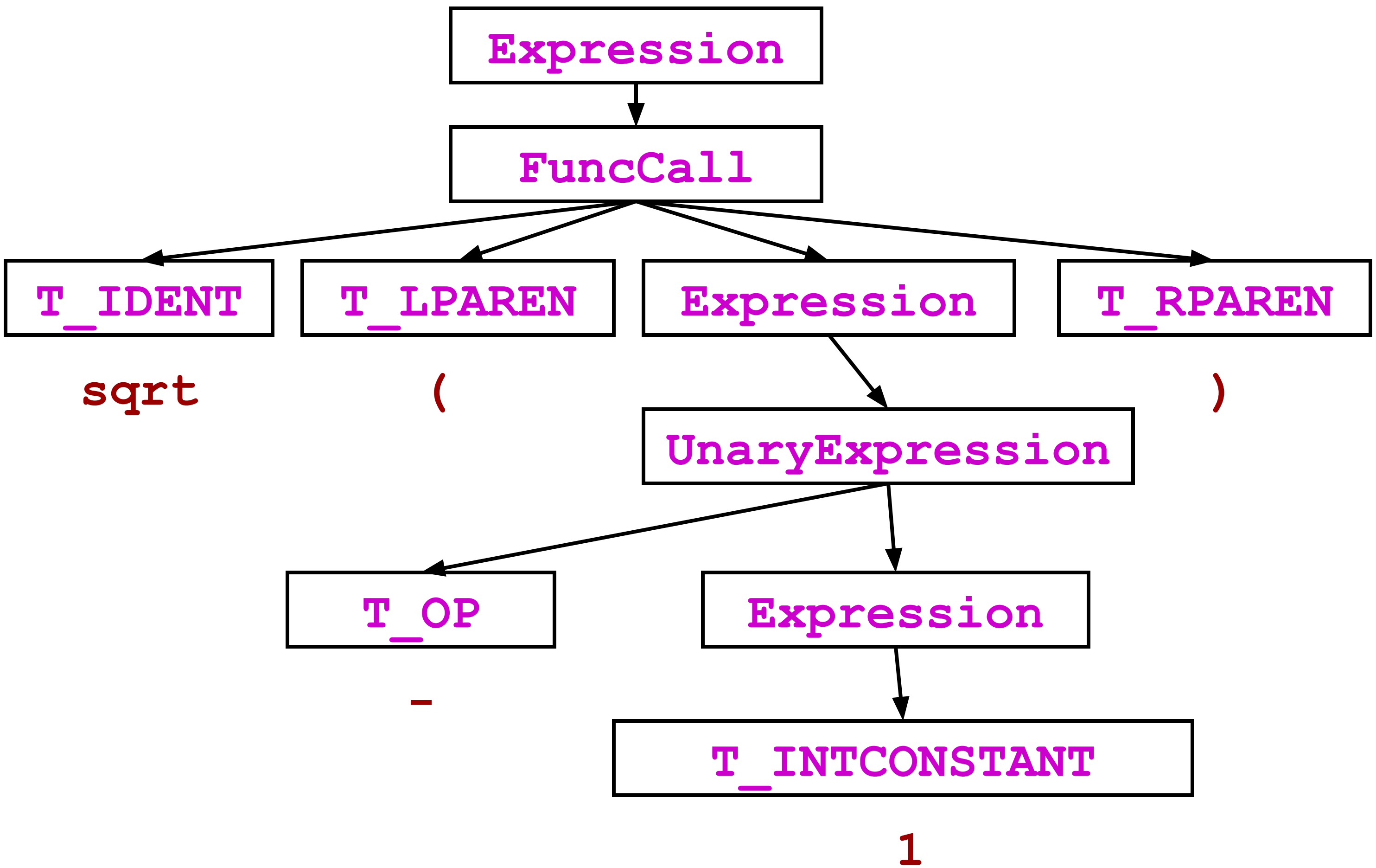
```
double f = sqrt(-1);
```

```
T_DOUBLE        ("double")
T_IDENT    ("f")
T_OP       ("=")
T_IDENT    ("sqrt")
T_LPAREN   ("(")
T_OP       ("-")
T_INTCONSTANT ("1")
T_RPAREN   (")")
T_SEP   (";")
```

Syntax Analysis

- Also called *parsing*
- Describe the set of strings that are programs using a grammar
- Structural validation
- Create a parse tree or derivation

Parse tree for `sqrt(-1)`



Abstract Syntax Tree

```
sqrt(-1):=
    MethodCall (
        sqrt,
        UnaryExpr(   UnaryMinus,
                     Number(1)
        )
    )
```

Semantic analysis

- "does it make sense"? Checking semantic rules,
  - Is there a `main` function?
  - Is variable declared?
  - Are operand types compatible? (coercion)
  - Do function arguments match function declarations?
- Type checking
- Static vs. run-time semantic checks
  - Array bounds, return values do not match definition

Compiler Back-end

Source -> abstract syntax tree

```
extern void print_int(int);

class C {
  bool foo() { return(true); }
  int main() {
    if (foo()) {
      print_int(1); }
    }
}
```

Source -> abstract syntax tree

```
Program(
    ExternFunction(print_int,VoidType,VarDef(IntType)),
    Class( C,
            None,
            Method(foo,
                    BoolType,
                    None,
                    MethodBlock( None,
                                 ReturnStmt(BoolExpr(True)))),
    Method(   main,
              IntType,
              None,
              MethodBlock( None,
                           IfStmt(MethodCall(foo,None),
                                  Block(None,
                                        MethodCall
                                    (print_int,Number(1))),
                           None)))))
```

Intermediate representation

```
; ModuleID = 'C'

declare void
@print_int(i32)

define i1 @foo() {
entry:
  ret i1 true
}
```

```
define i32 @main() {
entry:
  br label %ifstart
ifstart:
%calltmp = call i1 @foo()
  br i1 %calltmp, label %iftrue, label %end
iftrue:
call void @print_int(i32 1)
  br label %end
end:
  ret i32 0
}
```

Translation from IR to machine specific assembly

Intermediate representation

```
declare void @print_int(i32)
```

```
define i32 @main() {
entry:
  br label %ifstart
```

```
define i1 @foo() {
entry:
  ret i1 true
}
```

```
ifstart:
%calltmp = call i1 @foo()
  br i1 %calltmp, label %iftrue, label %end
```

```
iftrue:
call void @print_int(i32 1)
  br label %end
```

```
end:
  ret i32 0
}
```

Assembly language output from IR

x86 assembly

```
	.section	__TEXT,__text,
regular,pure_instructions
	.globl	_foo
	.align	4, 0x90
@foo
	.cfi_startproc
%entry
	mov	al, 1
	ret
	.cfi_endproc

	.globl	_main
	.align	4, 0x90
```

```
@main
	.cfi_startproc
%entry
	push	rax
Ltmp0:
	.cfi_def_cfa_offset 16
	call	_foo
	test	al, 1
	je	LBB1_2
%iftrue
	mov	edi, 1
	call	_print_int
%end
	xor	eax, eax
	pop	rdx
	ret
	.cfi_endproc
```

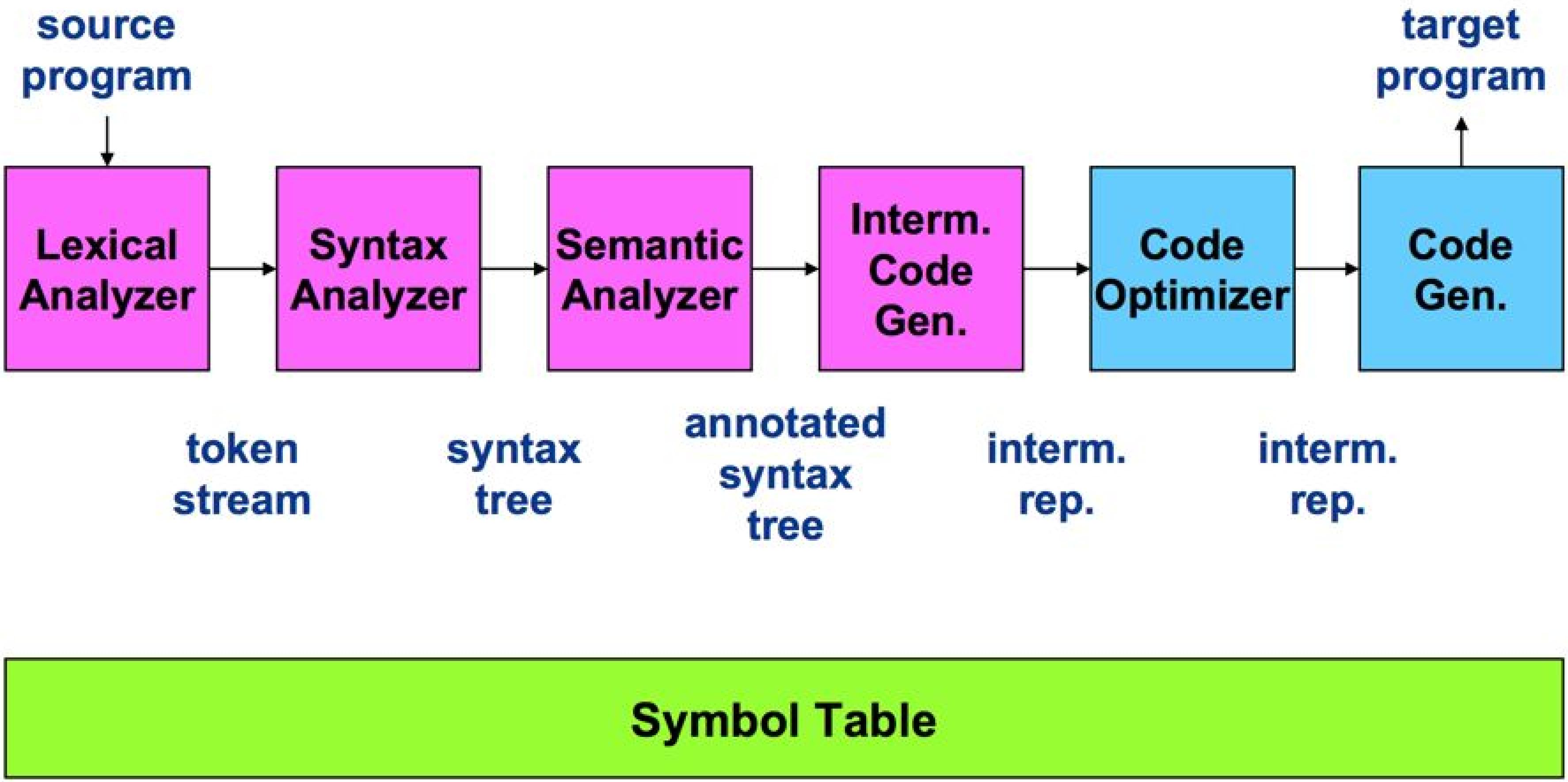
Code optimization

```
; ModuleID = 'C'

declare void @print_int(i32)

define i32 @main() {
entry:
  br label %ifstart

ifstart:
  call void @print_int(i32 1)
  br label %end

end:
  ret i32 0
}
```

Code Optimization


```
	.section	__TEXT,__text,regular,
pure_instructions
	.macosx_version_min 10, 11
	.globl	_main
	.p2align	4, 0x90
_main:
	.cfi_startproc
## BB#0:
	pushq	%rax
Ltmp0:
	.cfi_def_cfa_offset 16
	movl	$1, %edi
	callq	_print_int
	xorl	%eax, %eax
	popq	%rcx
	retq
	.cfi_endproc
```

Stages of a Compiler



Wrap Up

- Analysis/Synthesis
  - Translation from string to executable
- Divide and conquer
  - Build one component at a time
  - Theoretical analysis will ensure we keep things **simple** and **correct**
  - Create a complex piece of software