02247 Compiler Construction
Modules 1 & 2: Compilers overview & The j-- Compiler

Andrea Vandin
(*Slides based on: Swami Iyer, UMass Boston, http://www.swamiiyer.net/cs451/slides.html*)

Tentative Lecture Plan

#	*Date*	*Topic*	*Chapter*
1	Feb 1	Course Introduction & Compilers Overview	1
2	Feb 8	The *j*-- Compiler & Demo of Final Project	1
3	Feb 15	Lexical Analysis	2
4	Feb 22	Lexical Analysis 2	2
5	Mar 1	Lab Day	
6	Mar 8	Parsing	3
7	Mar 15	Parsing 2	3
8	Mar 22	Type Checking	4
	Mar 29	*Easter holiday*	
9	Apr 5	Type Checking 2	4
10	Apr 12	Lab Day	
11	Apr 19	JVM Code Generation	5
12	Apr 26	MIPS Code Generation	6
13	May 3	Register Allocation	7
	?	Translating functional languages	
	May 10	Deadline for final assignment	
	May 18	Exam	

Outline

1. **Compilers**
   - From Source Language to Target Language
   - Interpeters
   - Why Study Compilers?
2. **The Phases of Compilation**
   - Why modularize
   - Front End
   - Back End
   - Middle End: Optimizer
3. **Overview of the j-- to JVM Compiler**
   - The *j*-- language
   - Organization of the compiler
4. **The j-- Compiler Source Tree**
   - Naming conventions
   - The command-line interface of the *j*-- Compiler
   - Import the *j*-- Compiler in Eclipse
   - Extending the *j*-- Compiler: a demo of the final project
   - Extending the *j*-- Compiler: EXERCISE

Outline

1. **Compilers**
   - From Source Language to Target Language
   - Interpeters
   - Why Study Compilers?
2. **The Phases of Compilation**
   - Why modularize
   - Front End
   - Back End
   - Middle End: Optimizer
3. **Overview of the j-- to JVM Compiler**
   - The *j*-- language
   - Organization of the compiler
4. **The j-- Compiler Source Tree**
   - Naming conventions
   - The command-line interface of the *j*-- Compiler
   - Import the *j*-- Compiler in Eclipse
   - Extending the *j*-- Compiler: a demo of the final project
   - Extending the *j*-- Compiler: EXERCISE

What is a compiler?

What is a compiler?

A compiler translates a source program written in a high-level programming language (Java, C# or C), into a target program in a lower level language (machine code)

A programming language is an artificial language in which a programmer writes a program to control the behavior of a computer

Like a natural language, in a programming language one describes

- The tokens
  - *The admissible words*
- The syntax of programs and language constructs (classes, methods, statements, . . .)
  - *How to obtain legal sentences by combining words.*
- The semantics (aka meaning) of the various constructs
  - *The meaning of the sentences*

What is a compiler?

Source language

A compiler translates a source program written in a high-level programming language (Java, C# or C), into a target program in a lower level language (machine code)

A programming language is an artificial language in which a programmer writes a program to control the behavior of a computer

Like a natural language, in a programming language one describes

- The tokens
  - *The admissible words*
- The syntax of programs and language constructs (classes, methods, statements, . . .)
  - *How to obtain legal sentences by combining words.*
- The semantics (aka meaning) of the various constructs
  - *The meaning of the sentences*

What is a compiler?

Source language

A compiler translates a source program written in a high-level programming language (Java, C# or C), into a target program in a lower level language (machine code)

A programming language is an artificial language in which a programmer writes a program to control the behavior of a computer

Like a natural language, in a programming language one describes

- The tokens
  - *The admissible words*
- The syntax of programs and language constructs (classes, methods, statements, . . .)
  - *How to obtain legal sentences by combining words.*
- The semantics (aka meaning) of the various constructs
  - *The meaning of the sentences*

What is a compiler?

Source language

A compiler translates a source program written in a high-level programming language (Java, C# or C), into a target program in a lower level language (machine code)

A programming language is an artificial language in which a programmer writes a program to control the behavior of a computer

Like a natural language, in a programming language one describes

- The tokens
  - *The admissible words*
- The syntax of programs and language constructs (classes, methods, statements, . . .)
  - *How to obtain legal sentences by combining words.*
- The semantics (aka meaning) of the various constructs
  - *The meaning of the sentences*

What is a compiler?

Source language

A compiler translates a source program written in a high-level programming language (Java, C# or C), into a target program in a lower level language (machine code)

A programming language is an artificial language in which a programmer writes a program to control the behavior of a computer

Like a natural language, in a programming language one describes

- The tokens
  - *The admissible words*
- The syntax of programs and language constructs (classes, methods, statements, . . .)
  - *How to obtain legal sentences by combining words.*
- The semantics (aka meaning) of the various constructs
  - *The meaning of the sentences*

What is a compiler?

Target language

A compiler translates a source program written in a high-level programming language (Java, C# or C), into a target program in a lower level language (machine code)

The machine language of a computer is designed so as to be easily interpreted by the computer itself

A computer's machine language is also known as its instruction set or architecture.

- Intel i386, a complex instruction set computer (CISC) with powerful and complex instructions
- MIPS, a reduced instruction set computer (RISC) with relatively simple instructions
- Java Virtual Machine (JVM), a virtual machine not necessarily implemented in hardware

What is a compiler?

Target language

A compiler translates a source program written in a high-level programming language (Java, C# or C), into a target program in a lower level language (machine code)

The machine language of a computer is designed so as to be easily interpreted by the computer itself

A computer's machine language is also known as its instruction set or architecture.

- Intel i386, a complex instruction set computer (CISC) with powerful and complex instructions
- MIPS, a reduced instruction set computer (RISC) with relatively simple instructions
- Java Virtual Machine (JVM), a virtual machine not necessarily implemented in hardware

Outline

1. **Compilers**
   - From Source Language to Target Language
   - Interpeters
   - Why Study Compilers?
2. **The Phases of Compilation**
   - Why modularize
   - Front End
   - Back End
   - Middle End: Optimizer
3. **Overview of the j-- to JVM Compiler**
   - The *j*-- language
   - Organization of the compiler
4. **The j-- Compiler Source Tree**
   - Naming conventions
   - The command-line interface of the *j*-- Compiler
   - Import the *j*-- Compiler in Eclipse
   - Extending the *j*-- Compiler: a demo of the final project
   - Extending the *j*-- Compiler: EXERCISE

Compilers VS Interpreters

An interpreter executes a high-level language program directly, ie, the high-level program is first loaded into the interpreter, and then executed

Examples of programming languages whose programs may be interpreted directly are the UNIX shell languages, such as bash and csh, Python, and many versions of LISP

Compilers VS Interpreters

An interpreter executes a high-level language program directly, ie, the high-level program is first loaded into the interpreter, and then executed

Examples of programming languages whose programs may be interpreted directly are the UNIX shell languages, such as bash and csh, Python, and many versions of LISP

Outline

1. **Compilers**
   - From Source Language to Target Language
   - Interpeters
   - Why Study Compilers?
2. **The Phases of Compilation**
   - Why modularize
   - Front End
   - Back End
   - Middle End: Optimizer
3. **Overview of the j-- to JVM Compiler**
   - The *j*-- language
   - Organization of the compiler
4. **The j-- Compiler Source Tree**
   - Naming conventions
   - The command-line interface of the *j*-- Compiler
   - Import the *j*-- Compiler in Eclipse
   - Extending the *j*-- Compiler: a demo of the final project
   - Extending the *j*-- Compiler: EXERCISE

Why should we study compilers?

- Compilers are larger programs than those you have written/seen in your courses
- Compilers use many notions that you have learned in previous courses: arrays, lists, regular expressions, finite state automata, context-free grammars, recursion and patterns
- You learn a lot about the language you are compiling (in our case, Java)
- You learn a lot about the target machine (in our case, both the JVM and MIPS)
- Compilers are still being written for:
  - New languages and language updates
  - New computer architectures
- Some companies reimplemented compilers of existing languages (e.g., Java), to:
  - Implement specific security policies
  - Have the freedom to choose which updates to adopt
  - Have the freedom to modify the virtual machine (this is why .NET exists)
- There is a good mix of theory and practice, and each is relevant to the other
- Compilers are organized in separated stages, each building on the previous one
- Compiler writing is a case study in software engineering
- Compilers are programs and writing programs is fun

Why should we study compilers?

- Compilers are larger programs than those you have written/seen in your courses
- Compilers use many notions that you have learned in previous courses: arrays, lists, regular expressions, finite state automata, context-free grammars, recursion and patterns
- You learn a lot about the language you are compiling (in our case, Java)
- You learn a lot about the target machine (in our case, both the JVM and MIPS)
- Compilers are still being written for:
  - New languages and language updates
  - New computer architectures
- Some companies reimplemented compilers of existing languages (e.g., Java), to:
  - Implement specific security policies
  - Have the freedom to choose which updates to adopt
  - Have the freedom to modify the virtual machine (this is why .NET exists)
- There is a good mix of theory and practice, and each is relevant to the other
- Compilers are organized in separated stages, each building on the previous one
- Compiler writing is a case study in software engineering
- Compilers are programs and writing programs is fun

Why should we study compilers?

- Compilers are larger programs than those you have written/seen in your courses
- Compilers use many notions that you have learned in previous courses: arrays, lists, regular expressions, finite state automata, context-free grammars, recursion and patterns
- You learn a lot about the language you are compiling (in our case, Java)
- You learn a lot about the target machine (in our case, both the JVM and MIPS)
- Compilers are still being written for:
  - New languages and language updates
  - New computer architectures
- Some companies reimplemented compilers of existing languages (e.g., Java), to:
  - Implement specific security policies
  - Have the freedom to choose which updates to adopt
  - Have the freedom to modify the virtual machine (this is why .NET exists)
- There is a good mix of theory and practice, and each is relevant to the other
- Compilers are organized in separated stages, each building on the previous one
- Compiler writing is a case study in software engineering
- Compilers are programs and writing programs is fun

Why should we study compilers?

- Compilers are larger programs than those you have written/seen in your courses
- Compilers use many notions that you have learned in previous courses: arrays, lists, regular expressions, finite state automata, context-free grammars, recursion and patterns
- You learn a lot about the language you are compiling (in our case, Java)
- You learn a lot about the target machine (in our case, both the JVM and MIPS)
- Compilers are still being written for:
  - New languages and language updates
  - New computer architectures
- Some companies reimplemented compilers of existing languages (e.g., Java), to:
  - Implement specific security policies
  - Have the freedom to choose which updates to adopt
  - Have the freedom to modify the virtual machine (this is why .NET exists)
- There is a good mix of theory and practice, and each is relevant to the other
- Compilers are organized in separated stages, each building on the previous one
- Compiler writing is a case study in software engineering
- Compilers are programs and writing programs is fun

Why should we study compilers?

- Compilers are larger programs than those you have written/seen in your courses
- Compilers use many notions that you have learned in previous courses: arrays, lists, regular expressions, finite state automata, context-free grammars, recursion and patterns
- You learn a lot about the language you are compiling (in our case, Java)
- You learn a lot about the target machine (in our case, both the JVM and MIPS)
- Compilers are still being written for:
  - New languages and language updates
  - New computer architectures
- Some companies reimplemented compilers of existing languages (e.g., Java), to:
  - Implement specific security policies
  - Have the freedom to choose which updates to adopt
  - Have the freedom to modify the virtual machine (this is why .NET exists)
- There is a good mix of theory and practice, and each is relevant to the other
- Compilers are organized in separated stages, each building on the previous one
- Compiler writing is a case study in software engineering
- Compilers are programs and writing programs is fun

Why should we study compilers?

- Compilers are larger programs than those you have written/seen in your courses
- Compilers use many notions that you have learned in previous courses: arrays, lists, regular expressions, finite state automata, context-free grammars, recursion and patterns
- You learn a lot about the language you are compiling (in our case, Java)
- You learn a lot about the target machine (in our case, both the JVM and MIPS)
- Compilers are still being written for:
  - New languages and language updates
  - New computer architectures
- Some companies reimplemented compilers of existing languages (e.g., Java), to:
  - Implement specific security policies
  - Have the freedom to choose which updates to adopt
  - Have the freedom to modify the virtual machine (this is why .NET exists)
- There is a good mix of theory and practice, and each is relevant to the other
- Compilers are organized in separated stages, each building on the previous one
- Compiler writing is a case study in software engineering
- Compilers are programs and writing programs is fun

Why should we study compilers?

- Compilers are larger programs than those you have written/seen in your courses
- Compilers use many notions that you have learned in previous courses: arrays, lists, regular expressions, finite state automata, context-free grammars, recursion and patterns
- You learn a lot about the language you are compiling (in our case, Java)
- You learn a lot about the target machine (in our case, both the JVM and MIPS)
- Compilers are still being written for:
  - New languages and language updates
  - New computer architectures
- Some companies reimplemented compilers of existing languages (e.g., Java), to:
  - Implement specific security policies
  - Have the freedom to choose which updates to adopt
  - Have the freedom to modify the virtual machine (this is why .NET exists)
- There is a good mix of theory and practice, and each is relevant to the other
- Compilers are organized in separated stages, each building on the previous one
- Compiler writing is a case study in software engineering
- Compilers are programs and writing programs is fun

Why should we study compilers?

- Compilers are larger programs than those you have written/seen in your courses
- Compilers use many notions that you have learned in previous courses: arrays, lists, regular expressions, finite state automata, context-free grammars, recursion and patterns
- You learn a lot about the language you are compiling (in our case, Java)
- You learn a lot about the target machine (in our case, both the JVM and MIPS)
- Compilers are still being written for:
  - New languages and language updates
  - New computer architectures
- Some companies reimplemented compilers of existing languages (e.g., Java), to:
  - Implement specific security policies
  - Have the freedom to choose which updates to adopt
  - Have the freedom to modify the virtual machine (this is why .NET exists)
- There is a good mix of theory and practice, and each is relevant to the other
- Compilers are organized in separated stages, each building on the previous one
- Compiler writing is a case study in software engineering
- Compilers are programs and writing programs is fun

Why should we study compilers?

- Compilers are larger programs than those you have written/seen in your courses
- Compilers use many notions that you have learned in previous courses: arrays, lists, regular expressions, finite state automata, context-free grammars, recursion and patterns
- You learn a lot about the language you are compiling (in our case, Java)
- You learn a lot about the target machine (in our case, both the JVM and MIPS)
- Compilers are still being written for:
  - New languages and language updates
  - New computer architectures
- Some companies reimplemented compilers of existing languages (e.g., Java), to:
  - Implement specific security policies
  - Have the freedom to choose which updates to adopt
  - Have the freedom to modify the virtual machine (this is why .NET exists)
- There is a good mix of theory and practice, and each is relevant to the other
- Compilers are organized in separated stages, each building on the previous one
- Compiler writing is a case study in software engineering
- Compilers are programs and writing programs is fun

Why should we study compilers?

- Compilers are larger programs than those you have written/seen in your courses
- Compilers use many notions that you have learned in previous courses: arrays, lists, regular expressions, finite state automata, context-free grammars, recursion and patterns
- You learn a lot about the language you are compiling (in our case, Java)
- You learn a lot about the target machine (in our case, both the JVM and MIPS)
- Compilers are still being written for:
  - New languages and language updates
  - New computer architectures
- Some companies reimplemented compilers of existing languages (e.g., Java), to:
  - Implement specific security policies
  - Have the freedom to choose which updates to adopt
  - Have the freedom to modify the virtual machine (this is why .NET exists)
- There is a good mix of theory and practice, and each is relevant to the other
- Compilers are organized in separated stages, each building on the previous one
- Compiler writing is a case study in software engineering
- Compilers are programs and writing programs is fun

Outline

1. Compilers
   - From Source Language to Target Language
   - Interpeters
   - Why Study Compilers?
2. The Phases of Compilation
   - Why modularize
   - Front End
   - Back End
   - Middle End: Optimizer
3. Overview of the j-- to JVM Compiler
   - The *j*-- language
   - Organization of the compiler
4. The j-- Compiler Source Tree
   - Naming conventions
   - The command-line interface of the *j*-- Compiler
   - Import the *j*-- Compiler in Eclipse
   - Extending the *j*-- Compiler: a demo of the final project
   - Extending the *j*-- Compiler: EXERCISE

Compilers are organized in several phases/modules

Figure: Compiler as a black-box

The front end takes as input a high-level language program, and produces as output an intermediate representation (IR) of that program

- Source language dependent
- Target language independent

The back end then takes the IR of the program as input, and produces the target machine language program

- Source language independent
- Target language dependent

Compilers are organized in several phases/modules

A compiler can be broken into a **Front End** and a **Back End**

Figure: Compiler components: Front and Back End

The front end takes as input a high-level language program, and produces as output an intermediate representation (IR) of that program

- Source language dependent
- Target language independent

The back end then takes the IR of the program as input, and produces the target machine language program

- Source language independent
- Target language dependent

Compilers are organized in several phases/modules

A compiler can be broken into a Front End and a Back End

Figure: Compiler components: Front and Back End

The front end takes as input a high-level language program, and produces as output an intermediate representation (IR) of that program

- Source language dependent
- Target language independent

The back end then takes the IR of the program as input, and produces the target machine language program

- Source language independent
- Target language dependent

Compilers are organized in several phases/modules

A compiler can be broken into a **Front End** and a **Back End**

Figure: Compiler components: Front and Back End

The front end takes as input a high-level language program, and produces as output an intermediate representation (IR) of that program

- Source language dependent
- Target language independent

The back end then takes the IR of the program as input, and produces the target machine language program

- Source language independent
- Target language dependent

Compilers are organized in several phases/modules

WHY?

Compilers are organized in several phases/modules

WHY?

1. To reduce complexity: small programs are easier to understand (and implement)

Compilers are organized in several phases/modules

WHY?

1. To reduce complexity: small programs are easier to understand (and implement)
2. To reduce implementation time: more teams can work on different modules

Compilers are organized in several phases/modules

WHY?

1. To reduce complexity: small programs are easier to understand (and implement)
2. To reduce implementation time: more teams can work on different modules
3. To permit code re-use:

Compilers are organized in several phases/modules
WHY?

1. To reduce complexity: small programs are easier to understand (and implement)
2. To reduce implementation time: more teams can work on different modules
3. To permit code re-use:
   - ★ If you write a front end for C and a back end for the Intel Core Duo:
     - 1 compiler

Compilers are organized in several phases/modules
WHY?

1. To reduce complexity: small programs are easier to understand (and implement)
2. To reduce implementation time: more teams can work on different modules
3. To permit code re-use:
   - ★ If you write a front end for C and a back end for the Intel Core Duo:
     - 1 compiler
   - ★ If you add a new Java front end:
     - 2 compilers at the cost of 1.5

Compilers are organized in several phases/modules
WHY?

1. To reduce complexity: small programs are easier to understand (and implement)
2. To reduce implementation time: more teams can work on different modules
3. To permit code re-use:
   - ★ If you write a front end for C and a back end for the Intel Core Duo:
     - 1 compiler
   - ★ If you add a new Java front end:
     - 2 compilers at the cost of 1.5
   - ★ If you add a SPARC back end:
     - 4 compilers at the cost of 2

Outline

1. Compilers
   - From Source Language to Target Language
   - Interpeters
   - Why Study Compilers?
2. The Phases of Compilation
   - Why modularize
   - Front End
   - Back End
   - Middle End: Optimizer
3. Overview of the j-- to JVM Compiler
   - The *j*-- language
   - Organization of the compiler
4. The j-- Compiler Source Tree
   - Naming conventions
   - The command-line interface of the *j*-- Compiler
   - Import the *j*-- Compiler in Eclipse
   - Extending the *j*-- Compiler: a demo of the final project
   - Extending the *j*-- Compiler: EXERCISE

Front End phases

A compiler's front end analyzes the input program to determine its meaning

Figure: Compiler components: Front End

The front end can be further decomposed into a sequence of analysis phases

The scanner breaks the input stream of characters to a stream of tokens: identifiers, literals, reserved words, operators, and separators

The parser takes a sequence of tokens and parses it against a grammar to produce an abstract syntax tree (AST). This makes explicit the syntax in the source program

The semantics, or type checking, phase does semantic analysis: declares names in a symbol table, assigns types to expressions, and checks the validity of types

Front End phases

A compiler's front end analyzes the input program to determine its meaning

Figure: Compiler components: Front End

The front end can be further decomposed into a sequence of analysis phases

Figure: Front End phases

The scanner breaks the input stream of characters to a stream of tokens: identifiers, literals, reserved words, operators, and separators

The parser takes a sequence of tokens and parses it against a grammar to produce an abstract syntax tree (AST). This makes explicit the syntax in the source program

The semantics, or type checking, phase does semantic analysis: declares names in a symbol table, assigns types to expressions, and checks the validity of types

Front End phases

A compiler's front end analyzes the input program to determine its meaning

Figure: Compiler components: Front End

The front end can be further decomposed into a sequence of analysis phases

Figure: Front End phases

The scanner breaks the input stream of characters to a stream of tokens: identifiers, literals, reserved words, operators, and separators

The parser takes a sequence of tokens and parses it against a grammar to produce an abstract syntax tree (AST). This makes explicit the syntax in the source program

The semantics, or type checking, phase does semantic analysis: declares names in a symbol table, assigns types to expressions, and checks the validity of types

Front End phases

A compiler's front end analyzes the input program to determine its meaning

Figure: Compiler components: Front End

The front end can be further decomposed into a sequence of analysis phases

Figure: Front End phases

The scanner breaks the input stream of characters to a stream of tokens: identifiers, literals, reserved words, operators, and separators

The parser takes a sequence of tokens and parses it against a grammar to produce an abstract syntax tree (AST). This makes explicit the syntax in the source program

The semantics, or type checking, phase does semantic analysis: declares names in a symbol table, assigns types to expressions, and checks the validity of types

Front End phases

A compiler's front end analyzes the input program to determine its meaning

Figure: Compiler components: Front End

The front end can be further decomposed into a sequence of analysis phases

Figure: Front End phases

The scanner breaks the input stream of characters to a stream of tokens: identifiers, literals, reserved words, operators, and separators

The parser takes a sequence of tokens and parses it against a grammar to produce an abstract syntax tree (AST). This makes explicit the syntax in the source program

The semantics, or type checking, phase does semantic analysis: declares names in a symbol table, assigns types to expressions, and checks the validity of types

Front End phases

Example Input

```
float exp(float b, int n){
  // compute exp(b,n) naively
  // assuming n>=0
  float result = 1;
  while (n!=0){
    result = result * b;
    n = n-1;
  }
  return result;
}
```

Front End phases

Scanner (Lexical Analysis)

```
float exp ( float b , int n ) {
  // compute exp(b,n) naively
  // assuming n>=0
  float result = 1 ;
  while ( n != 0 ) {
    result = result * b ;
    n = n - 1 ;
  }
  return result ;
}
```

The scanner recognizes

- `Keywords` and `Symbols`
- User-defined `identifiers` for variables, functions, ...
- `Constants` like `0`
- Filters comments and whitespaces

Front End phases

Parser (Syntax Analysis)

```
float result = 1 ;
while ( n != 0 ) {
  result = result * b ;
  n = n - 1 ;
}
```

Front End phases

Type Checking (Semantic Analysis)

```
float result = 1 ;
while ( n != 0 ) {
  result = result * b ;
  n = n - 1 ;
}
```

Front End phases

Type Checking (Semantic Analysis)

```
float result = 1 ;
while ( n != 0 ) {
  result = result * b ;
  n = n - 1 ;
}
```

Outline

1 Compilers
- From Source Language to Target Language
- Interpeters
- Why Study Compilers?

2 The Phases of Compilation
- Why modularize
- Front End
- Back End
- Middle End: Optimizer

3 Overview of the j-- to JVM Compiler
- The *j*-- language
- Organization of the compiler

4 The j-- Compiler Source Tree
- Naming conventions
- The command-line interface of the *j*-- Compiler
- Import the *j*-- Compiler in Eclipse
- Extending the *j*-- Compiler: a demo of the final project
- Extending the *j*-- Compiler: EXERCISE

Back End phases

The back end takes the IR and produces a target machine program with same meaning

Figure: Compiler components: Back End

The back end can be further decomposed into a sequence of synthesis phases

The code generation phase generates machine instructions, making use of information collected in earlier phases

The peephole phase scans through the generated instructions looking locally for wasteful instruction sequences such as branches to branches and unnecessary load/store pairs

The object phase links together any modules produced in code generation and it constructs a single machine code executable program

Back End phases

The back end takes the IR and produces a target machine program with same meaning

Figure: Compiler components: Back End

The back end can be further decomposed into a sequence of synthesis phases

Figure: Back End phases

The code generation phase generates machine instructions, making use of information collected in earlier phases

The peephole phase scans through the generated instructions looking locally for wasteful instruction sequences such as branches to branches and unnecessary load/store pairs

The object phase links together any modules produced in code generation and it constructs a single machine code executable program

Back End phases

The back end takes the IR and produces a target machine program with same meaning

Figure: Compiler components: Back End

The back end can be further decomposed into a sequence of synthesis phases

Figure: Back End phases

The code generation phase generates machine instructions, making use of information collected in earlier phases

The peephole phase scans through the generated instructions looking locally for wasteful instruction sequences such as branches to branches and unnecessary load/store pairs

The object phase links together any modules produced in code generation and it constructs a single machine code executable program

Back End phases

The back end takes the IR and produces a target machine program with same meaning

Figure: Compiler components: Back End

The back end can be further decomposed into a sequence of synthesis phases

Figure: Back End phases

The code generation phase generates machine instructions, making use of information collected in earlier phases

The peephole phase scans through the generated instructions looking locally for wasteful instruction sequences such as branches to branches and unnecessary load/store pairs

The object phase links together any modules produced in code generation and it constructs a single machine code executable program

Back End phases

The back end takes the IR and produces a target machine program with same meaning

Figure: Compiler components: Back End

The back end can be further decomposed into a sequence of synthesis phases

Figure: Back End phases

The code generation phase generates machine instructions, making use of information collected in earlier phases

The peephole phase scans through the generated instructions looking locally for wasteful instruction sequences such as branches to branches and unnecessary load/store pairs

The object phase links together any modules produced in code generation and it constructs a single machine code executable program

Back End phases

Code Generation

```
        result:=1
loop:   if n compare.int 0 goto end
        result := result mult.float b
        n:= n sub.int 1
        goto loop
end:    ...
```

Back End phases

Code Generation – Register Allocation

```
b in r1, n in r2, result in r3, r4 empty
---------------------------------------
        r3:=1
loop:   if r2 compare.int 0 goto end
        r3 := r3 mult.float r1
        r2 := r2 sub.int 1
        goto loop
end:    ...
---------------------------------------
```

Back End phases

Peephole

```
b in r1, n in r2, result in r3, r4 empty
---------------------------------------------
            r3:=1
loop:       if r2 compare.int 0 goto end
            r3 := r3 mult.float r1
            goto asgn_r2
asgn_r2:    r2 := r2 sub.int 1
            goto loop
end:        ...
---------------------------------------------
```

Back End phases

Peephole

```
b in r1, n in r2, result in r3, r4 empty
---------------------------------------------
           r3:=1
loop:      if r2 compare.int 0 goto end
           r3 := r3 mult.float r1

asgn_r2:   r2 := r2 sub.int 1
           goto loop
end:       ...
---------------------------------------------
```

Outline

1 Compilers
- From Source Language to Target Language
- Interpeters
- Why Study Compilers?

2 The Phases of Compilation
- Why modularize
- Front End
- Back End
- Middle End: Optimizer

3 Overview of the j-- to JVM Compiler
- The *j*-- language
- Organization of the compiler

4 The j-- Compiler Source Tree
- Naming conventions
- The command-line interface of the *j*-- Compiler
- Import the *j*-- Compiler in Eclipse
- Extending the *j*-- Compiler: a demo of the final project
- Extending the *j*-- Compiler: EXERCISE

The Middle End: Optimizer

In some cases, a compiler can have a further phase named optimizer ("middle end")

The purpose of the optimizer is both to improve the IR program and to collect information that the back end may use for producing better code

The optimizer might:

- Compute next-use information for determining the lifetimes of variables, and gather loop information
- Next-use information is useful for eliminating common sub-expressions and constant folding (for example, replacing `x + 5` by `9` if we know `x` has the value `4`)
- Loop information is useful for pulling loop invariants out of loops and for strength reduction (for example, replacing multiplication operations by addition operations)

The Middle End: Optimizer

In some cases, a compiler can have a further phase named optimizer ("middle end")

The purpose of the optimizer is both to improve the IR program and to collect information that the back end may use for producing better code

The optimizer might:

- Compute next-use information for determining the lifetimes of variables, and gather loop information
- Next-use information is useful for eliminating common sub-expressions and constant folding (for example, replacing `x + 5` by `9` if we know `x` has the value `4`)
- Loop information is useful for pulling loop invariants out of loops and for strength reduction (for example, replacing multiplication operations by addition operations)

The Middle End: Optimizer

In some cases, a compiler can have a further phase named optimizer ("middle end")

The purpose of the optimizer is both to improve the IR program and to collect information that the back end may use for producing better code

The optimizer might:

- Compute next-use information for determining the lifetimes of variables, and gather loop information
- Next-use information is useful for eliminating common sub-expressions and constant folding (for example, replacing `x + 5` by `9` if we know `x` has the value `4`)
- Loop information is useful for pulling loop invariants out of loops and for strength reduction (for example, replacing multiplication operations by addition operations)

The Middle End: Optimizer

In some cases, a compiler can have a further phase named optimizer ("middle end")

The purpose of the optimizer is both to improve the IR program and to collect information that the back end may use for producing better code

The optimizer might:

- Compute next-use information for determining the lifetimes of variables, and gather loop information
- Next-use information is useful for eliminating common sub-expressions and constant folding (for example, replacing `x + 5` by `9` if we know `x` has the value `4`)
- Loop information is useful for pulling loop invariants out of loops and for strength reduction (for example, replacing multiplication operations by addition operations)

The Middle End: Optimizer

In some cases, a compiler can have a further phase named optimizer ("middle end")

The purpose of the optimizer is both to improve the IR program and to collect information that the back end may use for producing better code

The optimizer might:

- Compute next-use information for determining the lifetimes of variables, and gather loop information
- Next-use information is useful for eliminating common sub-expressions and constant folding (for example, replacing `x + 5` by `9` if we know `x` has the value `4`)
- Loop information is useful for pulling loop invariants out of loops and for strength reduction (for example, replacing multiplication operations by addition operations)

The Middle End: Optimizer

In some cases, a compiler can have a further phase named optimizer ("middle end")

The purpose of the optimizer is both to improve the IR program and to collect information that the back end may use for producing better code

The optimizer might:

- Compute next-use information for determining the lifetimes of variables, and gather loop information
- Next-use information is useful for eliminating common sub-expressions and constant folding (for example, replacing `x + 5` by `9` if we know `x` has the value `4`)
- Loop information is useful for pulling loop invariants out of loops and for strength reduction (for example, replacing multiplication operations by addition operations)

The Middle End: Optimizer

```
int n = 5;
int i = 0;
while(i <= n-1){
    int j = n * 2;
    System.out.println("i+j = " + (i + j));
    i++;
}
```

```
int n = 5;
int i = 0;
int j = n * 2;
while(i <= n-1){
    System.out.println("i+j = " + (i + j));
    i++;
}
```

```
int n = 5;
int i = 0;
int j = n + n;
while(i <= n-1){
    System.out.println("i+j = " + (i + j));
    i++;
}
```

```
int n = 5;
int i = 0;
int j = 10;
while(i <= 4){
    System.out.println("i+j = " + (i + j));
    i++;
}
```

The Middle End: Optimizer

```
int n = 5;
int i = 0;
while(i <= n-1){
    int j = n * 2;
    System.out.println("i+j = " + (i + j));
    i++;
}
```

```
int n = 5;
int i = 0;
int j = n * 2;
while(i <= n-1){
    System.out.println("i+j = " + (i + j));
    i++;
}
```

```
int n = 5;
int i = 0;
int j = n + n;
while(i <= n-1){
    System.out.println("i+j = " + (i + j));
    i++;
}
```

```
int n = 5;
int i = 0;
int j = 10;
while(i <= 4){
    System.out.println("i+j = " + (i + j));
    i++;
}
```

The Middle End: Optimizer

```
int n = 5;
int i = 0;
while(i <= n-1){
    int j = n * 2;
    System.out.println("i+j = " + (i + j));
    i++;
}
```

```
int n = 5;
int i = 0;
int j = n * 2;
while(i <= n-1){
    System.out.println("i+j = " + (i + j));
    i++;
}
```

```
int n = 5;
int i = 0;
int j = n + n;
while(i <= n-1){
    System.out.println("i+j = " + (i + j));
    i++;
}
```

```
int n = 5;
int i = 0;
int j = 10;
while(i <= 4){
    System.out.println("i+j = " + (i + j));
    i++;
}
```

The Middle End: Optimizer

```
int n = 5;
int i = 0;
while(i <= n-1){
    int j = n * 2;
    System.out.println("i+j = " + (i + j));
    i++;
}
```

```
int n = 5;
int i = 0;
int j = n * 2;
while(i <= n-1){
    System.out.println("i+j = " + (i + j));
    i++;
}
```

```
int n = 5;
int i = 0;
int j = n + n;
while(i <= n-1){
    System.out.println("i+j = " + (i + j));
    i++;
}
```

```
int n = 5;
int i = 0;
int j = 10;
while(i <= 4){
    System.out.println("i+j = " + (i + j));
    i++;
}
```

Outline

1. Compilers
   - From Source Language to Target Language
   - Interpeters
   - Why Study Compilers?
2. The Phases of Compilation
   - Why modularize
   - Front End
   - Back End
   - Middle End: Optimizer
3. Overview of the j-- to JVM Compiler
   - The *j*-- language
   - Organization of the compiler
4. The j-- Compiler Source Tree
   - Naming conventions
   - The command-line interface of the *j*-- Compiler
   - Import the *j*-- Compiler in Eclipse
   - Extending the *j*-- Compiler: a demo of the final project
   - Extending the *j*-- Compiler: EXERCISE

The j-- compiler

A simple Java-based compiler for a non-trivial subset of Java

The Java compiler, the program invoked when one types, for example

```
$ javac MyProgram.java
```

produces a `MyProgram.class` file, a byte code program suitable for execution on the JVM

To execute the `MyProgram.class` file, one types

```
$ java MyProgram
```

which effectively interprets the JVM program

In this course, we'll study the implementation of the *j*-- compiler, to compile a (non-trivial) subset of Java, which we also call *j*--

In the first instance, we'll target the JVM, which is a stack-based virtual machine

Then we will also study the compilation of JVM code to code for the MIPS machine, which is a register-based architecture; in doing this, we face the challenge of mapping possibly many variables to a limited number of fast registers

The j-- compiler

A simple Java-based compiler for a non-trivial subset of Java

The Java compiler, the program invoked when one types, for example

```
$ javac MyProgram.java
```

produces a `MyProgram.class` file, a byte code program suitable for execution on the JVM

To execute the `MyProgram.class` file, one types

```
$ java MyProgram
```

which effectively interprets the JVM program

In this course, we'll study the implementation of the *j*-- compiler, to compile a (non-trivial) subset of Java, which we also call *j*--

In the first instance, we'll target the JVM, which is a stack-based virtual machine

Then we will also study the compilation of JVM code to code for the MIPS machine, which is a register-based architecture; in doing this, we face the challenge of mapping possibly many variables to a limited number of fast registers

The j-- compiler

A simple Java-based compiler for a non-trivial subset of Java

The Java compiler, the program invoked when one types, for example

```
$ javac MyProgram.java
```

produces a `MyProgram.class` file, a byte code program suitable for execution on the JVM

To execute the `MyProgram.class` file, one types

```
$ java MyProgram
```

which effectively interprets the JVM program

In this course, we'll study the implementation of the *j*-- compiler, to compile a (non-trivial) subset of Java, which we also call *j*--

In the first instance, we'll target the JVM, which is a stack-based virtual machine

Then we will also study the compilation of JVM code to code for the MIPS machine, which is a register-based architecture; in doing this, we face the challenge of mapping possibly many variables to a limited number of fast registers

The j-- compiler

A simple Java-based compiler for a non-trivial subset of Java

The Java compiler, the program invoked when one types, for example

```
$ javac MyProgram.java
```

produces a `MyProgram.class` file, a byte code program suitable for execution on the JVM

To execute the `MyProgram.class` file, one types

```
$ java MyProgram
```

which effectively interprets the JVM program

In this course, we'll study the implementation of the *j*-- compiler, to compile a (non-trivial) subset of Java, which we also call *j*--

In the first instance, we'll target the JVM, which is a stack-based virtual machine

Then we will also study the compilation of JVM code to code for the MIPS machine, which is a register-based architecture; in doing this, we face the challenge of mapping possibly many variables to a limited number of fast registers

The j-- compiler

A simple Java-based compiler for a non-trivial subset of Java

The Java compiler, the program invoked when one types, for example

```
$ javac MyProgram.java
```

produces a `MyProgram.class` file, a byte code program suitable for execution on the JVM

To execute the `MyProgram.class` file, one types

```
$ java MyProgram
```

which effectively interprets the JVM program

In this course, we'll study the implementation of the *j*-- compiler, to compile a (non-trivial) subset of Java, which we also call *j*--

In the first instance, we'll target the JVM, which is a stack-based virtual machine

Then we will also study the compilation of JVM code to code for the MIPS machine, which is a register-based architecture; in doing this, we face the challenge of mapping possibly many variables to a limited number of fast registers

The j-- language

Our source language, *j*--, is a proper subset of the Java programming language:

- Roughly corresponds to Java 1.4
- Object-oriented programming language

Supports:

- classes, (static) methods, (static) fields
- inheritance, abstract classes
- casting
- if, if-else, while, blocks {}
- single-line comments
- int, boolean, char, String, arrays
- +, -, *

Does not support:

- interfaces, enumerations
- exceptions
- coercion (double = 5;)
- for
- multi-line comments
- byte, short, long, float, double
- /, %

The j-- language

Some examples of legal j-- programs

- HelloWorld
- Series
- Animalia

Outline

1 Compilers
- From Source Language to Target Language
- Interpeters
- Why Study Compilers?

2 The Phases of Compilation
- Why modularize
- Front End
- Back End
- Middle End: Optimizer

3 Overview of the j-- to JVM Compiler
- The *j*-- language
- Organization of the compiler

4 The j-- Compiler Source Tree
- Naming conventions
- The command-line interface of the *j*-- Compiler
- Import the *j*-- Compiler in Eclipse
- Extending the *j*-- Compiler: a demo of the final project
- Extending the *j*-- Compiler: EXERCISE

The organization of the j-- to JVM compiler

The *j*-- compiler is organized in an object-oriented fashion

The organization of the j-- to JVM compiler

The *j*-- compiler is organized in an object-oriented fashion

The organization of the j-- to JVM compiler

The *j*-- compiler is organized in an object-oriented fashion

The organization of the j-- to JVM compiler

The *j*-- compiler is organized in an object-oriented fashion

The organization of the j-- to JVM compiler

The *j*-- compiler is organized in an object-oriented fashion

The organization of the j-- to JVM compiler

The *j*-- compiler is organized in an object-oriented fashion

The organization of the j-- to JVM compiler

The scanner

The Scanner

The scanner supports the parser, scanning tokens from the input stream of characters comprising the source language program

For example, the scanner breaks down the following source language `HelloWorld` program

```
import java.lang.System;

public class HelloWorld {
    // The only method.
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

into atomic tokens such as and so on

Some tokens such as `java`, `HelloWorld`, and `main` are `IDENTIFIER` tokens , carrying along their images (string representations) as attributes

Some tokens such as `import`, `public`, and `class` are reserved words having the unique names such as `IMPORT`, `PUBLIC`, and `CLASS`

Operators and separators such as `+` and `;` have distinct names such as `PLUS` and `SEMI`

Literals such as `"Hello, World!"` comprise a single token such as `STRING_LITERAL`

Comments are scanned and ignored altogether

The Scanner

The scanner supports the parser, scanning tokens from the input stream of characters comprising the source language program

For example, the scanner breaks down the following source language `HelloWorld` program

```
import java.lang.System;

public class HelloWorld {
    // The only method.
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

into atomic tokens such as and so on

Some tokens such as `java`, `HelloWorld`, and `main` are `IDENTIFIER` tokens , carrying along their images (string representations) as attributes

Some tokens such as `import`, `public`, and `class` are reserved words having the unique names such as `IMPORT`, `PUBLIC`, and `CLASS`

Operators and separators such as `+` and `;` have distinct names such as `PLUS` and `SEMI`

Literals such as `"Hello, World!"` comprise a single token such as `STRING_LITERAL`

Comments are scanned and ignored altogether

The Scanner

The scanner supports the parser, scanning tokens from the input stream of characters comprising the source language program

For example, the scanner breaks down the following source language `HelloWorld` program

```
import java.lang.System;

public class HelloWorld {
    // The only method.
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

into atomic tokens such as `import`, `java`, `.`, `lang`, `.`, `System`, `;`, and so on

Some tokens such as `java`, `HelloWorld`, and `main` are `IDENTIFIER` tokens , carrying along their images (string representations) as attributes

Some tokens such as `import`, `public`, and `class` are reserved words having the unique names such as `IMPORT`, `PUBLIC`, and `CLASS`

Operators and separators such as `+` and `;` have distinct names such as `PLUS` and `SEMI`

Literals such as `"Hello, World!"` comprise a single token such as `STRING_LITERAL`

Comments are scanned and ignored altogether

The Scanner

The scanner supports the parser, scanning tokens from the input stream of characters comprising the source language program

For example, the scanner breaks down the following source language `HelloWorld` program

```
import java.lang.System;

public class HelloWorld {
    // The only method.
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

into atomic tokens such as `import`, `java`, `.`, `lang`, `.`, `System`, `;`, and so on

Some tokens such as `java`, `HelloWorld`, and `main` are `IDENTIFIER` tokens , carrying along their images (string representations) as attributes

Some tokens such as `import`, `public`, and `class` are reserved words having the unique names such as `IMPORT`, `PUBLIC`, and `CLASS`

Operators and separators such as `+` and `;` have distinct names such as `PLUS` and `SEMI`

Literals such as `"Hello, World!"` comprise a single token such as `STRING_LITERAL`

Comments are scanned and ignored altogether

The Scanner

The scanner supports the parser, scanning tokens from the input stream of characters comprising the source language program

For example, the scanner breaks down the following source language `HelloWorld` program

```
import java.lang.System;

public class HelloWorld {
    // The only method.
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

into atomic tokens such as `import`, `java`, `.`, `lang`, `.`, `System`, `;`, and so on

Some tokens such as `java`, `HelloWorld`, and `main` are `IDENTIFIER` tokens , carrying along their images (string representations) as attributes

Some tokens such as `import`, `public`, and `class` are reserved words having the unique names such as `IMPORT`, `PUBLIC`, and `CLASS`

Operators and separators such as `+` and `;` have distinct names such as `PLUS` and `SEMI`

Literals such as `"Hello, World!"` comprise a single token such as `STRING_LITERAL`

Comments are scanned and ignored altogether

The Scanner

The scanner supports the parser, scanning tokens from the input stream of characters comprising the source language program

For example, the scanner breaks down the following source language `HelloWorld` program

```
import java.lang.System;

public class HelloWorld {
    // The only method.
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

into atomic tokens such as `import`, `java`, `.`, `lang`, `.`, `System`, `;`, and so on

Some tokens such as `java`, `HelloWorld`, and `main` are `IDENTIFIER` tokens , carrying along their images (string representations) as attributes

Some tokens such as `import`, `public`, and `class` are reserved words having the unique names such as `IMPORT`, `PUBLIC`, and `CLASS`

Operators and separators such as `+` and `;` have distinct names such as `PLUS` and `SEMI`

Literals such as `"Hello, World!"` comprise a single token such as `STRING_LITERAL`

Comments are scanned and ignored altogether

The Scanner

The scanner supports the parser, scanning tokens from the input stream of characters comprising the source language program

For example, the scanner breaks down the following source language `HelloWorld` program

```
import java.lang.System;

public class HelloWorld {
    // The only method.
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

into atomic tokens such as `import`, `java`, `.`, `lang`, `.`, `System`, `;`, and so on

Some tokens such as `java`, `HelloWorld`, and `main` are `IDENTIFIER` tokens , carrying along their images (string representations) as attributes

Some tokens such as `import`, `public`, and `class` are reserved words having the unique names such as `IMPORT`, `PUBLIC`, and `CLASS`

Operators and separators such as `+` and `;` have distinct names such as `PLUS` and `SEMI`

Literals such as `"Hello, World!"` comprise a single token such as `STRING_LITERAL`

Comments are scanned and ignored altogether

The Scanner

The scanner supports the parser, scanning tokens from the input stream of characters comprising the source language program

For example, the scanner breaks down the following source language `HelloWorld` program

```
import java.lang.System;

public class HelloWorld {
    // The only method.
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

into atomic tokens such as `import`, `java`, `.`, `lang`, `.`, `System`, `;`, and so on

Some tokens such as `java`, `HelloWorld`, and `main` are `IDENTIFIER` tokens , carrying along their images (string representations) as attributes

Some tokens such as `import`, `public`, and `class` are reserved words having the unique names such as `IMPORT`, `PUBLIC`, and `CLASS`

Operators and separators such as `+` and `;` have distinct names such as `PLUS` and `SEMI`

Literals such as `"Hello, World!"` comprise a single token such as `STRING_LITERAL`

Comments are scanned and ignored altogether

The organization of the j-- to JVM compiler

The parser

Parsing

The parsing of a *j*-- program and the construction of its abstract syntax tree (AST) is driven by the language's syntax, and so is said to be syntax directed

In the first instance, our parser is hand-crafted from the *j*-- grammar, to parse *j*-- programs by a technique known as recursive descent

E.g., consider the grammatical rule describing the syntax for a compilation unit (a Java file)

```
compilationUnit ::= [PACKAGE qualifiedIdentifier SEMI]
                    {IMPORT  qualifiedIdentifier SEMI}
                    {typeDeclaration} EOF
```

This means that a compilation unit:

```
package pass;

import java.lang.System;

public class HelloWorld {
   // The only method.
   public static void main(String[] args) {
      ...
   }
}
```

Parsing

The parsing of a *j*-- program and the construction of its abstract syntax tree (AST) is driven by the language's syntax, and so is said to be syntax directed

In the first instance, our parser is hand-crafted from the *j*-- grammar, to parse *j*-- programs by a technique known as recursive descent

E.g., consider the grammatical rule describing the syntax for a compilation unit (a Java file)

```
compilationUnit ::= [PACKAGE qualifiedIdentifier SEMI]
                    {IMPORT  qualifiedIdentifier SEMI}
                    {typeDeclaration} EOF
```

This means that a compilation unit:

```
package pass;

import java.lang.System;

public class HelloWorld {
   // The only method.
   public static void main(String[] args) {
      ...
   }
}
```

Parsing

The parsing of a *j*-- program and the construction of its abstract syntax tree (AST) is driven by the language's syntax, and so is said to be syntax directed

In the first instance, our parser is hand-crafted from the *j*-- grammar, to parse *j*-- programs by a technique known as recursive descent

E.g., consider the grammatical rule describing the syntax for a compilation unit (a Java file)

```
compilationUnit ::= [PACKAGE qualifiedIdentifier SEMI]
                    {IMPORT  qualifiedIdentifier SEMI}
                    {typeDeclaration} EOF
```

This means that a compilation unit:

```
package pass;

import java.lang.System;

public class HelloWorld {
   // The only method.
   public static void main(String[] args) {
      ...
   }
}
```

Parsing

The parsing of a *j*-- program and the construction of its abstract syntax tree (AST) is driven by the language's syntax, and so is said to be syntax directed

In the first instance, our parser is hand-crafted from the *j*-- grammar, to parse *j*-- programs by a technique known as recursive descent

E.g., consider the grammatical rule describing the syntax for a compilation unit (a Java file)

```
compilationUnit ::= [PACKAGE qualifiedIdentifier SEMI]
                    {IMPORT  qualifiedIdentifier SEMI}
                    {typeDeclaration} EOF
```

This means that a compilation unit:

```
package pass;

import java.lang.System;

public class HelloWorld {
   // The only method.
   public static void main(String[] args) {
      ...
   }
}
```

Parsing

The parsing of a *j*-- program and the construction of its abstract syntax tree (AST) is driven by the language's syntax, and so is said to be syntax directed

In the first instance, our parser is hand-crafted from the *j*-- grammar, to parse *j*-- programs by a technique known as recursive descent

E.g., consider the grammatical rule describing the syntax for a compilation unit (a Java file)

```
compilationUnit ::= [PACKAGE qualifiedIdentifier SEMI]
                    {IMPORT  qualifiedIdentifier SEMI}
                    {typeDeclaration} EOF
```

This means that a compilation unit:

- Can begin with an optional [·] package declaration

```
package pass;

import java.lang.System;

public class HelloWorld {
   // The only method.
   public static void main(String[] args) {
      ...
   }
}
```

Parsing

The parsing of a *j*-- program and the construction of its abstract syntax tree (AST) is driven by the language's syntax, and so is said to be syntax directed

In the first instance, our parser is hand-crafted from the *j*-- grammar, to parse *j*-- programs by a technique known as recursive descent

E.g., consider the grammatical rule describing the syntax for a compilation unit (a Java file)

```
compilationUnit ::= [PACKAGE qualifiedIdentifier SEMI]
                    {IMPORT  qualifiedIdentifier SEMI}
                    {typeDeclaration} EOF
```

This means that a compilation unit:

- Can begin with an optional [·] package declaration
- Then it can have 0,1, or more {·} imports

```
package pass;

import java.lang.System;

public class HelloWorld {
   // The only method.
   public static void main(String[] args) {
      ...
   }
}
```

Parsing

The parsing of a *j*-- program and the construction of its abstract syntax tree (AST) is driven by the language's syntax, and so is said to be syntax directed

In the first instance, our parser is hand-crafted from the *j*-- grammar, to parse *j*-- programs by a technique known as recursive descent

E.g., consider the grammatical rule describing the syntax for a compilation unit (a Java file)

```
compilationUnit ::= [PACKAGE qualifiedIdentifier SEMI]
                    {IMPORT  qualifiedIdentifier SEMI}
                    {typeDeclaration} EOF
```

This means that a compilation unit:

- Can begin with an optional $[\cdot]$ package declaration
- Then it can have 0,1, or more $\{\cdot\}$ imports
- And it can declare 0,1, or more types (classes, enum, etc)

```java
package pass;

import java.lang.System;

public class HelloWorld {
   // The only method.
   public static void main(String[] args) {
      ...
   }
}
```

Parsing

The parsing of a *j*-- program and the construction of its abstract syntax tree (AST) is driven by the language's syntax, and so is said to be syntax directed

In the first instance, our parser is hand-crafted from the *j*-- grammar, to parse *j*-- programs by a technique known as recursive descent

E.g., consider the grammatical rule describing the syntax for a compilation unit (a Java file)

```
compilationUnit ::= [PACKAGE qualifiedIdentifier SEMI]
                    {IMPORT  qualifiedIdentifier SEMI}
                    {typeDeclaration} EOF
```

To parse a compilation unit using the recursive descent technique, one would write a method, call it `compilationUnit()`

```
    public JCompilationUnit compilationUnit() {
        ...
    }
```

Parsing

The parsing of a *j*-- program and the construction of its abstract syntax tree (AST) is driven by the language's syntax, and so is said to be syntax directed

In the first instance, our parser is hand-crafted from the *j*-- grammar, to parse *j*-- programs by a technique known as recursive descent

E.g., consider the grammatical rule describing the syntax for a compilation unit (a Java file)

```
compilationUnit ::= [PACKAGE qualifiedIdentifier SEMI]
                    {IMPORT  qualifiedIdentifier SEMI}
                    {typeDeclaration} EOF
```

To parse a compilation unit using the recursive descent technique, one would write a method, call it `compilationUnit()`

```
    public JCompilationUnit compilationUnit() {
          ...









    }
```

Parsing

The parsing of a *j*-- program and the construction of its abstract syntax tree (AST) is driven by the language's syntax, and so is said to be syntax directed

In the first instance, our parser is hand-crafted from the *j*-- grammar, to parse *j*-- programs by a technique known as recursive descent

E.g., consider the grammatical rule describing the syntax for a compilation unit (a Java file)

```
compilationUnit ::= [PACKAGE qualifiedIdentifier SEMI]
                    {IMPORT  qualifiedIdentifier SEMI}
                    {typeDeclaration} EOF
```

To parse a compilation unit using the recursive descent technique, one would write a method, call it `compilationUnit()`

```
    public JCompilationUnit compilationUiit() {
        if (have(PACKAGE)) {
            Read and store the package name...
        }
        ...
    }
```

Parsing

The parsing of a *j*-- program and the construction of its abstract syntax tree (AST) is driven by the language's syntax, and so is said to be syntax directed

In the first instance, our parser is hand-crafted from the *j*-- grammar, to parse *j*-- programs by a technique known as recursive descent

E.g., consider the grammatical rule describing the syntax for a compilation unit (a Java file)

```
compilationUnit ::= [PACKAGE qualifiedIdentifier SEMI]
                    {IMPORT  qualifiedIdentifier SEMI}
                    {typeDeclaration} EOF
```

To parse a compilation unit using the recursive descent technique, one would write a method, call it `compilationUnit()`

```
    public JCompilationUnit compilationUnit() {
        if (have(PACKAGE)) {
            Read and store the package name...
        }
        while (have(IMPORT)) {
            Read and store all imports...
        }
        ...




    }
```

Parsing

The parsing of a *j*-- program and the construction of its abstract syntax tree (AST) is driven by the language's syntax, and so is said to be syntax directed

In the first instance, our parser is hand-crafted from the *j*-- grammar, to parse *j*-- programs by a technique known as recursive descent

E.g., consider the grammatical rule describing the syntax for a compilation unit (a Java file)

```
compilationUnit ::= [PACKAGE qualifiedIdentifier SEMI]
                    {IMPORT  qualifiedIdentifier SEMI}
                    {typeDeclaration} EOF
```

To parse a compilation unit using the recursive descent technique, one would write a method, call it `compilationUnit()`

```
    public JCompilationUnit compilationUnit() {
        if (have(PACKAGE)) {
            Read and store the package name...
        }
        while (have(IMPORT)) {
            Read and store all imports...
        }
        while (!see(EOF)) {
           Read and store all type declarations...
        }
        mustBe(EOF);
        return new JCompilationUnit(stored_information)
    }
```

Parsing

To parse a compilation unit using the recursive descent technique, one would write a method, call it `compilationUnit()`

```
public JCompilationUnit compilationUnit() {
    int line = scanner.token().line();
    TypeName packageName = null; // Default
    //If the next token is 'package'
    if (have(PACKAGE)) {
        //Read the package name
        packageName = qualifiedIdentifier();
        //The package name must be followed by ';'
        mustBe(SEMI);
    }

    ArrayList<TypeName> imports = new ArrayList<TypeName>();
    while (have(IMPORT)) {
        imports.add(qualifiedIdentifier());
        mustBe(SEMI);
    }
    ArrayList<JAST> typeDeclarations = new ArrayList<JAST>();
    while (!see(EOF)) {
        JAST typeDeclaration = typeDeclaration();
        if (typeDeclaration != null) {
            typeDeclarations.add(typeDeclaration);
        }
    }
    mustBe(EOF);
    return new JCompilationUnit(scanner.fileName(), line, packageName,
            imports, typeDeclarations);
}
```

Parsing: An AST for the `HelloWorld` program

```
package pass;

import java.lang.System;

public class HelloWorld {
   // The only method.
   public static void main(String[] args) {
      System.out.println("Hello, World!");
   }
}
```

Parsing: An AST for the `HelloWorld` program

```
package pass;
```

Parsing: An AST for the `HelloWorld` program

```
package pass;

import java.lang.System;
```

Parsing: An AST for the `HelloWorld` program

```
package pass;

import java.lang.System;

public class HelloWorld {
```

Parsing: An AST for the `HelloWorld` program

```
package pass;

import java.lang.System;

public class HelloWorld {
   // The only method.
   public static void main(String[] args) {
```

Parsing: An AST for the `HelloWorld` program

```
package pass;

import java.lang.System;

public class HelloWorld {
   // The only method.
   public static void main(String[] args) {
      System.out.println("Hello, World!");
   }
}
```

The organization of the j-- to JVM compiler

Type checking: `preAnalyze()` and `analyze()`

Type checking: `preAnalyze()` and `analyze()`

As in Java, *j*-- names and values have types, and since *j*-- (like Java) is statically typed, its compiler must determine the types of all names and expressions

Types in *j*-- are represented using: `Type` (wraps `java.lang.Class`), `Method` (wraps `java.lang.reflect.Method`), `Constructor` (wraps `java.lang.reflect.Constructor`), `Field` (wraps `java.lang.reflect.Field`), and `Member` (wraps `java.lang.reflect.Member`)

There are places where we want to denote a type by its name, before its type is known or defined, and for this we use `TypeName` and (because we need array types) `ArrayTypeName`

The *j*-- compiler maintains a symbol table (a singly-linked list of `Context` objects) in which it declares names; each object in the symbol table represents some area of scope and contains a mapping from names to definitions:

- A `CompilationUnitContext` object represents the scope comprising the entire compilation unit
- A `ClassContext` object represents the scope of a class declaration
- A `LocalContext` object represents the scope of a block
- A `MethodContext` (subclass of `LocalContext`) object represents the scopes of methods and, by extension, constructors

Type checking: `preAnalyze()` and `analyze()`

As in Java, *j*-- names and values have types, and since *j*-- (like Java) is statically typed, its compiler must determine the types of all names and expressions

Types in *j*-- are represented using: `Type` (wraps `java.lang.Class`), `Method` (wraps `java.lang.reflect.Method`), `Constructor` (wraps `java.lang.reflect.Constructor`), `Field` (wraps `java.lang.reflect.Field`), and `Member` (wraps `java.lang.reflect.Member`)

There are places where we want to denote a type by its name, before its type is known or defined, and for this we use `TypeName` and (because we need array types) `ArrayTypeName`

The *j*-- compiler maintains a symbol table (a singly-linked list of `Context` objects) in which it declares names; each object in the symbol table represents some area of scope and contains a mapping from names to definitions:

- A `CompilationUnitContext` object represents the scope comprising the entire compilation unit
- A `ClassContext` object represents the scope of a class declaration
- A `LocalContext` object represents the scope of a block
- A `MethodContext` (subclass of `LocalContext`) object represents the scopes of methods and, by extension, constructors

Type checking: `preAnalyze()` and `analyze()`

As in Java, *j*-- names and values have types, and since *j*-- (like Java) is statically typed, its compiler must determine the types of all names and expressions

Types in *j*-- are represented using: `Type` (wraps `java.lang.Class`), `Method` (wraps `java.lang.reflect.Method`), `Constructor` (wraps `java.lang.reflect.Constructor`), `Field` (wraps `java.lang.reflect.Field`), and `Member` (wraps `java.lang.reflect.Member`)

There are places where we want to denote a type by its name, before its type is known or defined, and for this we use `TypeName` and (because we need array types) `ArrayTypeName`

The *j*-- compiler maintains a symbol table (a singly-linked list of `Context` objects) in which it declares names; each object in the symbol table represents some area of scope and contains a mapping from names to definitions:

- A `CompilationUnitContext` object represents the scope comprising the entire compilation unit
- A `ClassContext` object represents the scope of a class declaration
- A `LocalContext` object represents the scope of a block
- A `MethodContext` (subclass of `LocalContext`) object represents the scopes of methods and, by extension, constructors

Type checking: `preAnalyze()` and `analyze()`

As in Java, *j*-- names and values have types, and since *j*-- (like Java) is statically typed, its compiler must determine the types of all names and expressions

Types in *j*-- are represented using: `Type` (wraps `java.lang.Class`), `Method` (wraps `java.lang.reflect.Method`), `Constructor` (wraps `java.lang.reflect.Constructor`), `Field` (wraps `java.lang.reflect.Field`), and `Member` (wraps `java.lang.reflect.Member`)

There are places where we want to denote a type by its name, before its type is known or defined, and for this we use `TypeName` and (because we need array types) `ArrayTypeName`

The *j*-- compiler maintains a symbol table (a singly-linked list of `Context` objects) in which it declares names; each object in the symbol table represents some area of scope and contains a mapping from names to definitions:

- A `CompilationUnitContext` object represents the scope comprising the entire compilation unit
- A `ClassContext` object represents the scope of a class declaration
- A `LocalContext` object represents the scope of a block
- A `MethodContext` (subclass of `LocalContext`) object represents the scopes of methods and, by extension, constructors

Type checking: `preAnalyze()` and `analyze()`

As in Java, *j*-- names and values have types, and since *j*-- (like Java) is statically typed, its compiler must determine the types of all names and expressions

Types in *j*-- are represented using: `Type` (wraps `java.lang.Class`), `Method` (wraps `java.lang.reflect.Method`), `Constructor` (wraps `java.lang.reflect.Constructor`), `Field` (wraps `java.lang.reflect.Field`), and `Member` (wraps `java.lang.reflect.Member`)

There are places where we want to denote a type by its name, before its type is known or defined, and for this we use `TypeName` and (because we need array types) `ArrayTypeName`

The *j*-- compiler maintains a symbol table (a singly-linked list of `Context` objects) in which it declares names; each object in the symbol table represents some area of scope and contains a mapping from names to definitions:

- A `CompilationUnitContext` object represents the scope comprising the entire compilation unit
- A `ClassContext` object represents the scope of a class declaration
- A `LocalContext` object represents the scope of a block
- A `MethodContext` (subclass of `LocalContext`) object represents the scopes of methods and, by extension, constructors

Type checking: `preAnalyze()` and `analyze()`

As in Java, *j*-- names and values have types, and since *j*-- (like Java) is statically typed, its compiler must determine the types of all names and expressions

Types in *j*-- are represented using: `Type` (wraps `java.lang.Class`), `Method` (wraps `java.lang.reflect.Method`), `Constructor` (wraps `java.lang.reflect.Constructor`), `Field` (wraps `java.lang.reflect.Field`), and `Member` (wraps `java.lang.reflect.Member`)

There are places where we want to denote a type by its name, before its type is known or defined, and for this we use `TypeName` and (because we need array types) `ArrayTypeName`

The *j*-- compiler maintains a symbol table (a singly-linked list of `Context` objects) in which it declares names; each object in the symbol table represents some area of scope and contains a mapping from names to definitions:

- A `CompilationUnitContext` object represents the scope comprising the entire compilation unit
- A `ClassContext` object represents the scope of a class declaration
- A `LocalContext` object represents the scope of a block
- A `MethodContext` (subclass of `LocalContext`) object represents the scopes of methods and, by extension, constructors

Type checking: `preAnalyze()` and `analyze()`

As in Java, *j*-- names and values have types, and since *j*-- (like Java) is statically typed, its compiler must determine the types of all names and expressions

Types in *j*-- are represented using: `Type` (wraps `java.lang.Class`), `Method` (wraps `java.lang.reflect.Method`), `Constructor` (wraps `java.lang.reflect.Constructor`), `Field` (wraps `java.lang.reflect.Field`), and `Member` (wraps `java.lang.reflect.Member`)

There are places where we want to denote a type by its name, before its type is known or defined, and for this we use `TypeName` and (because we need array types) `ArrayTypeName`

The *j*-- compiler maintains a symbol table (a singly-linked list of `Context` objects) in which it declares names; each object in the symbol table represents some area of scope and contains a mapping from names to definitions:

- A `CompilationUnitContext` object represents the scope comprising the entire compilation unit
- A `ClassContext` object represents the scope of a class declaration
- A `LocalContext` object represents the scope of a block
- A `MethodContext` (subclass of `LocalContext`) object represents the scopes of methods and, by extension, constructors

Type checking: `preAnalyze()` and `analyze()`

As in Java, *j*-- names and values have types, and since *j*-- (like Java) is statically typed, its compiler must determine the types of all names and expressions

Types in *j*-- are represented using: `Type` (wraps `java.lang.Class`), `Method` (wraps `java.lang.reflect.Method`), `Constructor` (wraps `java.lang.reflect.Constructor`), `Field` (wraps `java.lang.reflect.Field`), and `Member` (wraps `java.lang.reflect.Member`)

There are places where we want to denote a type by its name, before its type is known or defined, and for this we use `TypeName` and (because we need array types) `ArrayTypeName`

The *j*-- compiler maintains a symbol table (a singly-linked list of `Context` objects) in which it declares names; each object in the symbol table represents some area of scope and contains a mapping from names to definitions:

- A `CompilationUnitContext` object represents the scope comprising the entire compilation unit
- A `ClassContext` object represents the scope of a class declaration
- A `LocalContext` object represents the scope of a block
- A `MethodContext` (subclass of `LocalContext`) object represents the scopes of methods and, by extension, constructors

Type checking: `preAnalyze()` and `analyze()`

`preAnalyze()` is a first pass at type checking, which builds that part of the symbol table that is at the top of the AST, to declare both imported types and types introduced by class declarations, and to declare the members in those classes

`analyze()` picks up where `preAnalyze()` left off and continues to build the symbol table, decorating the AST with type information and enforcing the *j*-- type rules

`analyze()` also performs other important tasks such as type checking, accessibility, member finding, tree rewriting, and storage allocation

Type checking: `preAnalyze()` and `analyze()`

`preAnalyze()` is a first pass at type checking, which builds that part of the symbol table that is at the top of the AST, to declare both imported types and types introduced by class declarations, and to declare the members in those classes

`analyze()` picks up where `preAnalyze()` left off and continues to build the symbol table, decorating the AST with type information and enforcing the *j*-- type rules

`analyze()` also performs other important tasks such as type checking, accessibility, member finding, tree rewriting, and storage allocation

Type checking: `preAnalyze()` and `analyze()`

`preAnalyze()` is a first pass at type checking, which builds that part of the symbol table that is at the top of the AST, to declare both imported types and types introduced by class declarations, and to declare the members in those classes

`analyze()` picks up where `preAnalyze()` left off and continues to build the symbol table, decorating the AST with type information and enforcing the *j*-- type rules

`analyze()` also performs other important tasks such as type checking, accessibility, member finding, tree rewriting, and storage allocation

The organization of the j-- to JVM compiler

JVM Code Generation: `codegen()`

JVM Code Generation: `codegen()`

The JVM is a stack machine: all computations are carried out atop the run-time stack

Each time a method is invoked, the JVM:

- allocates a stack frame, a contiguous block of memory locations on top of the run-time stack;
- assigns the method arguments, the values of local variables, and the temporary results to positions within this stack frame

JVM Code Generation: `codegen()`

The JVM is a stack machine: all computations are carried out atop the run-time stack

Each time a method is invoked, the JVM:

- allocates a stack frame, a contiguous block of memory locations on top of the run-time stack;
- assigns the method arguments, the values of local variables, and the temporary results to positions within this stack frame

Stack Frame for a Static Method

Stack Frame for an Instance Method

JVM Code Generation: `codegen()`

The purpose of `codegen()` is to generate JVM byte code from the AST, based on information computed by `preAnalyze()` and `analyze()`

`codegen()` is invoked by `Main` on the root of the AST, and `codegen()` recursively descends the AST, generating byte code

`CLEmitter` provides an abstraction for the JVM class file, hiding many of the gory details

Here's the implementation of `codegen()` in `JMethodDeclaration` (AST representation for method declarations)

```
public void codegen(CLEmitter output) {
    output.addMethod(mods, name, descriptor, null, false);
    if (body != null) {
        body.codegen(output);
    }

    // Add implicit RETURN
    if (returnType == Type.VOID) {
        output.addNoArgInstruction(RETURN);
    }
}
```

JVM Code Generation: `codegen()`

The purpose of `codegen()` is to generate JVM byte code from the AST, based on information computed by `preAnalyze()` and `analyze()`

`codegen()` is invoked by `Main` on the root of the AST, and `codegen()` recursively descends the AST, generating byte code

`CLEmitter` provides an abstraction for the JVM class file, hiding many of the gory details

Here's the implementation of `codegen()` in `JMethodDeclaration` (AST representation for method declarations)

```
public void codegen(CLEmitter output) {
    output.addMethod(mods, name, descriptor, null, false);
    if (body != null) {
        body.codegen(output);
    }

    // Add implicit RETURN
    if (returnType == Type.VOID) {
        output.addNoArgInstruction(RETURN);
    }
}
```

JVM Code Generation: `codegen()`

The purpose of `codegen()` is to generate JVM byte code from the AST, based on information computed by `preAnalyze()` and `analyze()`

`codegen()` is invoked by `Main` on the root of the AST, and `codegen()` recursively descends the AST, generating byte code

`CLEmitter` provides an abstraction for the JVM class file, hiding many of the gory details

Here's the implementation of `codegen()` in `JMethodDeclaration` (AST representation for method declarations)

```
public void codegen(CLEmitter output) {
    output.addMethod(mods, name, descriptor, null, false);
    if (body != null) {
        body.codegen(output);
    }

    // Add implicit RETURN
    if (returnType == Type.VOID) {
        output.addNoArgInstruction(RETURN);
    }
}
```

JVM Code Generation: `codegen()`

The purpose of `codegen()` is to generate JVM byte code from the AST, based on information computed by `preAnalyze()` and `analyze()`

`codegen()` is invoked by `Main` on the root of the AST, and `codegen()` recursively descends the AST, generating byte code

`CLEmitter` provides an abstraction for the JVM class file, hiding many of the gory details

Here's the implementation of `codegen()` in `JMethodDeclaration` (AST representation for method declarations)

```
public void codegen(CLEmitter output) {
    output.addMethod(mods, name, descriptor, null, false);
    if (body != null) {
        body.codegen(output);
    }

    // Add implicit RETURN
    if (returnType == Type.VOID) {
        output.addNoArgInstruction(RETURN);
    }
}
```

Outline

1. Compilers
   - From Source Language to Target Language
   - Interpeters
   - Why Study Compilers?
2. The Phases of Compilation
   - Why modularize
   - Front End
   - Back End
   - Middle End: Optimizer
3. Overview of the j-- to JVM Compiler
   - The *j*-- language
   - Organization of the compiler
4. The j-- Compiler Source Tree
   - Naming conventions
   - The command-line interface of the *j*-- Compiler
   - Import the *j*-- Compiler in Eclipse
   - Extending the *j*-- Compiler: a demo of the final project
   - Extending the *j*-- Compiler: EXERCISE

The Source Tree of the j-- Compiler

The zip file `j--.zip` containing the *j*-- distribution may be unzipped into any directory of your choosing.

`j--/src/jminusminus` contains the source files for the compiler, where `jminusminus` is a package, and these include

- `Main.java`, the driver program
- A hand-written scanner (`Scanner.java`) and parser (`Parser.java`)
- `J*.java` files defining classes representing the AST nodes
- `CL*.java` files supplying the back-end code that is used by *j*-- for creating JVM byte code; the most important file amongst these is `CLEmitter.java`, which provides the interface between the front end and back end of the compiler
- `S*.java` files that translate JVM code to SPIM files (SPIM is an interpreter for the MIPS machines symbolic assembly language)
- `j--.jj`, the input file to JavaCC containing the specification for generating (as opposed to hand-writing) a scanner and parser for the *j*-- language; `JavaCCMain`, the driver program that uses the scanner and parser produced by JavaCC
- Other Java files providing representation for types and the symbol table

The Source Tree of the j-- Compiler

The zip file `j--.zip` containing the *j*-- distribution may be unzipped into any directory of your choosing.

`j--/src/jminusminus` contains the source files for the compiler, where `jminusminus` is a package, and these include

- `Main.java`, the driver program
- A hand-written scanner (`Scanner.java`) and parser (`Parser.java`)
- `J*.java` files defining classes representing the AST nodes
- `CL*.java` files supplying the back-end code that is used by *j*-- for creating JVM byte code; the most important file amongst these is `CLEmitter.java`, which provides the interface between the front end and back end of the compiler
- `S*.java` files that translate JVM code to SPIM files (SPIM is an interpreter for the MIPS machines symbolic assembly language)
- `j--.jj`, the input file to JavaCC containing the specification for generating (as opposed to hand-writing) a scanner and parser for the *j*-- language; `JavaCCMain`, the driver program that uses the scanner and parser produced by JavaCC
- Other Java files providing representation for types and the symbol table

The Source Tree of the j-- Compiler

The zip file `j--.zip` containing the *j*-- distribution may be unzipped into any directory of your choosing.

`j--/src/jminusminus` contains the source files for the compiler, where `jminusminus` is a package, and these include

- `Main.java`, the driver program
- A hand-written scanner (`Scanner.java`) and parser (`Parser.java`)
- `J*.java` files defining classes representing the AST nodes
- `CL*.java` files supplying the back-end code that is used by *j*-- for creating JVM byte code; the most important file amongst these is `CLEmitter.java`, which provides the interface between the front end and back end of the compiler
- `S*.java` files that translate JVM code to SPIM files (SPIM is an interpreter for the MIPS machines symbolic assembly language)
- `j--.jj`, the input file to JavaCC containing the specification for generating (as opposed to hand-writing) a scanner and parser for the *j*-- language; `JavaCCMain`, the driver program that uses the scanner and parser produced by JavaCC
- Other Java files providing representation for types and the symbol table

The Source Tree of the j-- Compiler

The zip file `j--.zip` containing the *j*-- distribution may be unzipped into any directory of your choosing.

`j--/src/jminusminus` contains the source files for the compiler, where `jminusminus` is a package, and these include

- `Main.java`, the driver program
- A hand-written scanner (`Scanner.java`) and parser (`Parser.java`)
- `J*.java` files defining classes representing the AST nodes
- `CL*.java` files supplying the back-end code that is used by *j*-- for creating JVM byte code; the most important file amongst these is `CLEmitter.java`, which provides the interface between the front end and back end of the compiler
- `S*.java` files that translate JVM code to SPIM files (SPIM is an interpreter for the MIPS machines symbolic assembly language)
- `j--.jj`, the input file to JavaCC containing the specification for generating (as opposed to hand-writing) a scanner and parser for the *j*-- language; `JavaCCMain`, the driver program that uses the scanner and parser produced by JavaCC
- Other Java files providing representation for types and the symbol table

The Source Tree of the j-- Compiler

The zip file `j--.zip` containing the *j*-- distribution may be unzipped into any directory of your choosing.

`j--/src/jminusminus` contains the source files for the compiler, where `jminusminus` is a package, and these include

- `Main.java`, the driver program
- A hand-written scanner (`Scanner.java`) and parser (`Parser.java`)
- `J*.java` files defining classes representing the AST nodes
- `CL*.java` files supplying the back-end code that is used by *j*-- for creating JVM byte code; the most important file amongst these is `CLEmitter.java`, which provides the interface between the front end and back end of the compiler
- `S*.java` files that translate JVM code to SPIM files (SPIM is an interpreter for the MIPS machines symbolic assembly language)
- `j--.jj`, the input file to JavaCC containing the specification for generating (as opposed to hand-writing) a scanner and parser for the *j*-- language; `JavaCCMain`, the driver program that uses the scanner and parser produced by JavaCC
- Other Java files providing representation for types and the symbol table

The Source Tree of the j-- Compiler

The zip file `j--.zip` containing the *j*-- distribution may be unzipped into any directory of your choosing.

`j--/src/jminusminus` contains the source files for the compiler, where `jminusminus` is a package, and these include

- `Main.java`, the driver program
- A hand-written scanner (`Scanner.java`) and parser (`Parser.java`)
- `J*.java` files defining classes representing the AST nodes
- `CL*.java` files supplying the back-end code that is used by *j*-- for creating JVM byte code; the most important file amongst these is `CLEmitter.java`, which provides the interface between the front end and back end of the compiler
- `S*.java` files that translate JVM code to SPIM files (SPIM is an interpreter for the MIPS machines symbolic assembly language)
- `j--.jj`, the input file to JavaCC containing the specification for generating (as opposed to hand-writing) a scanner and parser for the *j*-- language; `JavaCCMain`, the driver program that uses the scanner and parser produced by JavaCC
- Other Java files providing representation for types and the symbol table

The Source Tree of the j-- Compiler

The zip file `j--.zip` containing the *j*-- distribution may be unzipped into any directory of your choosing.

`j--/src/jminusminus` contains the source files for the compiler, where `jminusminus` is a package, and these include

- `Main.java`, the driver program
- A hand-written scanner (`Scanner.java`) and parser (`Parser.java`)
- `J*.java` files defining classes representing the AST nodes
- `CL*.java` files supplying the back-end code that is used by *j*-- for creating JVM byte code; the most important file amongst these is `CLEmitter.java`, which provides the interface between the front end and back end of the compiler
- `S*.java` files that translate JVM code to SPIM files (SPIM is an interpreter for the MIPS machines symbolic assembly language)
- `j--.jj`, the input file to JavaCC containing the specification for generating (as opposed to hand-writing) a scanner and parser for the *j*-- language; `JavaCCMain`, the driver program that uses the scanner and parser produced by JavaCC
- Other Java files providing representation for types and the symbol table

The Source Tree of the j-- Compiler

The zip file `j--.zip` containing the *j*-- distribution may be unzipped into any directory of your choosing.

`j--/src/jminusminus` contains the source files for the compiler, where `jminusminus` is a package, and these include

- `Main.java`, the driver program
- A hand-written scanner (`Scanner.java`) and parser (`Parser.java`)
- `J*.java` files defining classes representing the AST nodes
- `CL*.java` files supplying the back-end code that is used by *j*-- for creating JVM byte code; the most important file amongst these is `CLEmitter.java`, which provides the interface between the front end and back end of the compiler
- `S*.java` files that translate JVM code to SPIM files (SPIM is an interpreter for the MIPS machines symbolic assembly language)
- `j--.jj`, the input file to JavaCC containing the specification for generating (as opposed to hand-writing) a scanner and parser for the *j*-- language; `JavaCCMain`, the driver program that uses the scanner and parser produced by JavaCC
- Other Java files providing representation for types and the symbol table

Outline

1. Compilers
   - From Source Language to Target Language
   - Interpeters
   - Why Study Compilers?
2. The Phases of Compilation
   - Why modularize
   - Front End
   - Back End
   - Middle End: Optimizer
3. Overview of the j-- to JVM Compiler
   - The *j*-- language
   - Organization of the compiler
4. The j-- Compiler Source Tree
   - Naming conventions
   - The command-line interface of the *j*-- Compiler
   - Import the *j*-- Compiler in Eclipse
   - Extending the *j*-- Compiler: a demo of the final project
   - Extending the *j*-- Compiler: EXERCISE

The command-line interface of the j-- Compiler

`j--/bin/j--` is a script to run the compiler and has the following command-line syntax

```
Usage: j-- <options> <source file>
where possible options include:
  -t Only tokenize input and print tokens to STDOUT
  -p Only parse input and print AST to STDOUT
  -pa Only parse and pre-analyze input and print AST to STDOUT
  -a Only parse, pre-analyze, and analyze input and print AST to STDOUT
  -s <naive|linear|graph> Generate SPIM code
  -r <num> Max. physical registers (1-18) available for allocation; default = 8
  -d <dir> Specify where to place output files; default = .
```

For example, the *j*-- program `j--/tests/pass/HelloWorld.java` can be compiled using *j*-- as follows

```
$ j--/bin/j-- $j/j--/tests/pass/HelloWorld.java
```

to produce a `HelloWorld.class` file under `pass` folder within the current directory, which can then be run as

```
$ java pass.HelloWorld
```

to produce as output

```
$ Hello, World!
```

The command-line interface of the j-- Compiler

`j--/bin/j--` is a script to run the compiler and has the following command-line syntax

```
Usage: j-- <options> <source file>
where possible options include:
  -t Only tokenize input and print tokens to STDOUT
  -p Only parse input and print AST to STDOUT
  -pa Only parse and pre-analyze input and print AST to STDOUT
  -a Only parse, pre-analyze, and analyze input and print AST to STDOUT
  -s <naive|linear|graph> Generate SPIM code
  -r <num> Max. physical registers (1-18) available for allocation; default = 8
  -d <dir> Specify where to place output files; default = .
```

For example, the *j*-- program `j--/tests/pass/HelloWorld.java` can be compiled using *j*-- as follows

```
$ j--/bin/j-- $j/j--/tests/pass/HelloWorld.java
```

to produce a `HelloWorld.class` file under `pass` folder within the current directory, which can then be run as

```
$ java pass.HelloWorld
```

to produce as output

```
$ Hello, World!
```

Outline

1. Compilers
   - From Source Language to Target Language
   - Interpeters
   - Why Study Compilers?
2. The Phases of Compilation
   - Why modularize
   - Front End
   - Back End
   - Middle End: Optimizer
3. Overview of the j-- to JVM Compiler
   - The *j*-- language
   - Organization of the compiler
4. The j-- Compiler Source Tree
   - Naming conventions
   - The command-line interface of the *j*-- Compiler
   - Import the *j*-- Compiler in Eclipse
   - Extending the *j*-- Compiler: a demo of the final project
   - Extending the *j*-- Compiler: EXERCISE

Demo: j-- in Eclipse

Import the source code of the j-- Compiler in Eclipse

1. Unzip *j*--.zip
2. Open Eclipse, and create a new Java project named *j*--
3. Right-click on the project in the package explorer, then click *Import* → *General* → *File System* → *Next*
4. Click on *Browse...* to locate the folder obtained from *j*--.zip
5. Select j– on the left to select all files in the folder, and then deselect .project.
6. Click *Finish*, and then *yes to all* if prompted
7. Repeat steps 3-6, but this time in step 5 deselect all files but .project. Click *Finish*

Disable the automatic build feature (if you prefer)

- Deselect *Build Automatically* in the menu *Project*.
- Build the *j*-- manually by selecting the project in the Project Explorer, and clicking the *Project* → *Build Project* menu. This will also run all tests.

Run the compiler

- Click on *Run* → *Run Configurations...*
- Select *Java Application* and press the *New* button
- Set j– launcher as *Name*, and Main - jminusminus as *Main class*

Outline

1. Compilers
   - From Source Language to Target Language
   - Interpeters
   - Why Study Compilers?
2. The Phases of Compilation
   - Why modularize
   - Front End
   - Back End
   - Middle End: Optimizer
3. Overview of the j-- to JVM Compiler
   - The *j*-- language
   - Organization of the compiler
4. The j-- Compiler Source Tree
   - Naming conventions
   - The command-line interface of the *j*-- Compiler
   - Import the *j*-- Compiler in Eclipse
   - Extending the *j*-- Compiler: a demo of the final project
   - Extending the *j*-- Compiler: EXERCISE

Extending the j-- Compiler: a demo of the final assignment

j-- provides an elaborate framework with which one may add new Java constructs to the *j*-- language

As an illustrative example, let's add the division operator to j--, which involves:

- modifying the scanner to recognize `/` as a token,
- modifying the parser to be able to parse division expressions,
- implementing semantic analysis,
- implementing code generation for the division operation

Extending the j-- Compiler: a demo of the final assignment

j-- provides an elaborate framework with which one may add new Java constructs to the *j*-- language

As an illustrative example, let's add the division operator to j--, which involves:

- modifying the scanner to recognize `/` as a token,
- modifying the parser to be able to parse division expressions,
- implementing semantic analysis,
- implementing code generation for the division operation

Extending the j-- Compiler: a demo of the final assignment

Writing tests

- Place the following file (`Division.java`) under `j--/tests/pass`

```
package pass;

public class Division {
    public int divide(int x, int y) {
        return x / y ;
    }
}
```

This will be used twice . . .

Extending the j-- Compiler: a demo of the final assignment

Writing tests

- Place the following file (`Division.java`) under `j--/tests/pass`

```
package pass;

public class Division {
    public int divide(int x, int y) {
        return x / y ;
    }
}
```

This will be used twice . . .

Extending the j-- Compiler: a demo of the final assignment

Writing tests

- Place the following file (`DivisionTest.java`) under `j--/tests/junit`

```
package junit;

import junit.framework.TestCase;
import pass.Division;

public class DivisionTest extends TestCase {
    private Division division;

    protected void setUp() throws Exception {
        super.setUp();
        division = new Division();
    }

    protected void tearDown() throws Exception {
        super.tearDown();
    }

    public void testDivide() {
        this.assertEquals(division.divide(0, 42), 0);
        this.assertEquals(division.divide(42, 1), 42);
        this.assertEquals(division.divide(127, 3), 42);
    }
}
```

- Make the following entry in the `suite()` method of `junit.JMinusMinusTestRunner`

```
TestSuite suite = new TestSuite();
...
suite.addTestSuite(DivisionTest.class);
return suite;
```

Extending the j-- Compiler: a demo of the final assignment

Writing tests

- Place the following file (`DivisionTest.java`) under `j--/tests/junit`

```
package junit;

import junit.framework.TestCase;
import pass.Division;

public class DivisionTest extends TestCase {
    private Division division;

    protected void setUp() throws Exception {
        super.setUp();
        division = new Division();
    }

    protected void tearDown() throws Exception {
        super.tearDown();
    }

    public void testDivide() {
        this.assertEquals(division.divide(0, 42), 0);
        this.assertEquals(division.divide(42, 1), 42);
        this.assertEquals(division.divide(127, 3), 42);
    }
}
```

- Make the following entry in the `suite()` method of `junit.JMinusMinusTestRunner`

```
TestSuite suite = new TestSuite();
...
suite.addTestSuite(DivisionTest.class);
return suite;
```

Extending the j-- Compiler: a demo of the final assignment

Write tests

- Place the following file (`Division.java`) under `j--/tests/fail`

```
package fail;

import java.lang.System;

public class Division {
    public static void main(String[] args) {
        System.out.println('a' / 42);
    }
}
```

Extending the j-- Compiler: a demo of the final assignment

Changes to lexical and syntactic grammars

- Add a line describing the division operator to `j--/lexicalgrammar` under the operators section

```
DIV ::= "/"
```

- Add a rule for the division operator to the rule describing multiplicative expressions in the `j--/grammar` file

```
multiplicativeExpression ::= unaryExpression // level 2
                               {(STAR | DIV) unaryExpression}
```

Extending the j-- Compiler: a demo of the final assignment

Changes to scanner

- Make the following change in `TokenInfo.java` under `j--/src/jminusminus`

```
enum TokenKind {
    EOF (" < EOF >") ,
    ... ,
    STAR ("*") ,
    DIV ("/") ,
    ...
}
```

- Make the following change in `Scanner.java` under `j--/src/jminusminus`

```
if (ch == '/'   ) {
    nextCh();
    if (ch == '/') {
        // CharReader maps all new lines to '\n'
        while (ch != '\n' && ch != EOFCH) {
            nextCh();
        }
    }
    else {
        return new TokenInfo(DIV, line);
    }
}
```

Extending the j-- Compiler: a demo of the final assignment

Changes to scanner

- Make the following change in `TokenInfo.java` under `j--/src/jminusminus`

```
enum TokenKind {
    EOF (" < EOF >") ,
    ... ,
    STAR ("*") ,
    DIV ("/") ,
    ...
}
```

- Make the following change in `Scanner.java` under `j--/src/jminusminus`

```
if (ch == '/'   ) {
    nextCh();
    if (ch == '/') {
        // CharReader maps all new lines to '\n'
        while (ch != '\n' && ch != EOFCH) {
            nextCh();
        }
    }
    else {
        return new TokenInfo(DIV, line);
    }
}
```

Extending the j-- Compiler: a demo of the final assignment

Changes to scanner

- Make the following change in `TokenInfo.java` under `j--/src/jminusminus`

```
enum TokenKind {
    EOF (" < EOF >") ,
    ... ,
    STAR ("*") ,
    DIV ("/") ,
    ...
}
```

- Make the following change in `Scanner.java` under `j--/src/jminusminus`

```
if (ch == '/'   ) {
    nextCh();
    if (ch == '/') {
        // CharReader maps all new lines to '\n'
        while (ch != '\n' && ch != EOFCH) {
            nextCh();
        }
    }
    else {
        return new TokenInfo(DIV, line);
    }
}
```

Test the extension of the scanner using: `-t tests/pass/Division.java`

Extending the j-- Compiler: a demo of the final assignment

Changes to parser

- Define class `JDivideOp` in `JBinaryExpression.java` to represent an AST node for the `/` operator

```
class JDivideOp extends JBinaryExpression {
    public JDivideOp(int line, JExpression lhs, JExpression rhs) {
        super(line, "/", lhs, rhs);
    }

    public JExpression analyze (Context context) { return this ; }

    public void codegen(CLEmitter output) { }
}
```

- To parse expressions involving division operator, modify the `multiplicativeExpression()` method in `Parser.java` as follows

```
private JExpression multiplicativeExpression() {
    int line = scanner.token().line();
    boolean more = true;
    JExpression lhs = unaryExpression();
    while (more) {
        if (have(STAR)) {
            lhs = new JMultiplyOp(line, lhs, unaryExpression());
        }
        else if (have(DIV)) {
            lhs = new JDivideOp(line, lhs, unaryExpression());
        }
        else { more = false; }
    }
    return lhs;
}
```

Extending the j-- Compiler: a demo of the final assignment

Changes to parser

- Define class `JDivideOp` in `JBinaryExpression.java` to represent an AST node for the `/` operator

```
class JDivideOp extends JBinaryExpression {
    public JDivideOp(int line, JExpression lhs, JExpression rhs) {
        super(line, "/", lhs, rhs);
    }

    public JExpression analyze (Context context) { return this ; }

    public void codegen(CLEmitter output) { }
}
```

- To parse expressions involving division operator, modify the `multiplicativeExpression()` method in `Parser.java` as follows

```
private JExpression multiplicativeExpression() {
    int line = scanner.token().line();
    boolean more = true;
    JExpression lhs = unaryExpression();
    while (more) {
        if (have(STAR)) {
            lhs = new JMultiplyOp(line, lhs, unaryExpression());
        }
        else if (have(DIV)) {
            lhs = new JDivideOp(line, lhs, unaryExpression());
        }
        else { more = false; }
    }
    return lhs;
}
```

Extending the j-- Compiler: a demo of the final assignment

Changes to parser

- Define class `JDivideOp` in `JBinaryExpression.java` to represent an AST node for the `/` operator

```
class JDivideOp extends JBinaryExpression {
    public JDivideOp(int line, JExpression lhs, JExpression rhs) {
        super(line, "/", lhs, rhs);
    }

    public JExpression analyze (Context context) { return this ; }

    public void codegen(CLEmitter output) { }
}
```

- To parse expressions involving division operator, modify the `multiplicativeExpression()` method in `Parser.java` as follows

```
private JExpression multiplicativeExpression() {
    int line = scanner.token().line();
    boolean more = true;
    JExpression lhs = unaryExpression();
    while (more) {
        if (have(STAR)) {
            lhs = new JMultiplyOp(line, lhs, unaryExpression());
        }
        else if (have(DIV)) {
            lhs = new JDivideOp(line, lhs, unaryExpression());
        }
        else { more = false; }
    }
    return lhs;
}
```

Test the extension of the parser using: `-p tests/pass/Division.java`

Extending the j-- Compiler: a demo of the final assignment

Semantic analysis and code generation

- Implement `analyze()` in `JDivideOp` as follows

```
public JExpression analyze(Context context) {
    lhs = (JExpression) lhs.analyze(context);
    rhs = (JExpression) rhs.analyze(context);
    lhs.type().mustMatchExpected(line(), Type.INT);
    rhs.type().mustMatchExpected(line(), Type.INT);
    type = Type.INT;
    return this;
}
```

- Implement `codegen()` in `JDivideOp` as follows

```
public void codegen(CLEmitter output) {
    lhs.codegen(output);
    rhs.codegen(output);
    output.addNoArgInstruction(IDIV);
}
```

Test the extension:

- build the Eclipse project.
- `-d outputfolder tests/pass/Division.java`
- Write your own *j*-- file, and compile it

Extending the j-- Compiler: a demo of the final assignment

Semantic analysis and code generation

- Implement `analyze()` in `JDivideOp` as follows

```
public JExpression analyze(Context context) {
    lhs = (JExpression) lhs.analyze(context);
    rhs = (JExpression) rhs.analyze(context);
    lhs.type().mustMatchExpected(line(), Type.INT);
    rhs.type().mustMatchExpected(line(), Type.INT);
    type = Type.INT;
    return this;
}
```

- Implement `codegen()` in `JDivideOp` as follows

```
public void codegen(CLEmitter output) {
    lhs.codegen(output);
    rhs.codegen(output);
    output.addNoArgInstruction(IDIV);
}
```

Test the extension:

- build the Eclipse project.
- `-d outputfolder tests/pass/Division.java`
- Write your own *j*-- file, and compile it

Extending the j-- Compiler: a demo of the final assignment

Semantic analysis and code generation

- Implement `analyze()` in `JDivideOp` as follows

```
public JExpression analyze(Context context) {
    lhs = (JExpression) lhs.analyze(context);
    rhs = (JExpression) rhs.analyze(context);
    lhs.type().mustMatchExpected(line(), Type.INT);
    rhs.type().mustMatchExpected(line(), Type.INT);
    type = Type.INT;
    return this;
}
```

- Implement `codegen()` in `JDivideOp` as follows

```
public void codegen(CLEmitter output) {
    lhs.codegen(output);
    rhs.codegen(output);
    output.addNoArgInstruction(IDIV);
}
```

Test the extension:

- build the Eclipse project.
- `-d outputfolder tests/pass/Division.java`
- Write your own *j*-- file, and compile it

Extending the j-- Compiler: EXERCISE

Your task is to add the remainder (%) operator to j--, which involves:

- modifying the scanner to recognize % as a token,
- modifying the parser to be able to parse remainder expressions,
- implementing semantic analysis,
- implementing code generation for the remainder operation

Extending the j-- Compiler: EXERCISE

Writing tests

- Place the following file (`Remainder.java`) under `j--/tests/pass`

```
package pass;

public class Remainder {
    public int remainder(int x, int y) {
        return x % y ;
    }
}
```

- Place the following file (`Remainder.java`) under `j--/tests/fail`

```
package fail;

import java.lang.System;

public class Remainder {
    public static void main(String[] args) {
        System.out.println('a' % 42);
    }
}
```

Extending the j-- Compiler: EXERCISE

Writing tests

- Place the following file (`RemainderTest.java`) under `j--/tests/junit`

```
package junit;

import junit.framework.TestCase;
import pass.Remainder;

public class RemainderTest extends TestCase {
    private Remainder remainder;

    protected void setUp() throws Exception {
        super.setUp();
        remainder = new Remainder();
    }

    protected void tearDown() throws Exception {
        super.tearDown();
    }

    public void testRemainder() {
        this.assertEquals(remainder.remainder(0, 42), 0);
        this.assertEquals(remainder.remainder(42, 1), 0);
        this.assertEquals(remainder.remainder(127, 3), 1);
    }
}
```

- Make the following entry in the `suite()` method of `junit.JMinusMinusTestRunner`

```
TestSuite suite = new TestSuite();
...
suite.addTestSuite(RemainderTest.class);
return suite;
```

Extending the j-- Compiler: EXERCISE

Changes to lexical and syntactic grammars

- Add a line describing the remainder operator to `j--/lexicalgrammar` under the operators section

```
REM ::= "%"
```

- Add a rule for the division operator to the rule describing multiplicative expressions in the `j--/grammar` file

```
multiplicativeExpression ::= unaryExpression // level 2
                              {(STAR | DIV | REM) unaryExpression}
```

Extending the j-- Compiler: EXERCISE

The rest is up to you . . .

- Changes to the scanner
- Changes to the parser
- Semantic analysis
- Code generation