02247 Compiler Construction
Project Description

Andrea Vandin

Outline

Step 0: Get familiar with j--

- Description: Add basic Java operations on primitive integers to the *j--* language
- Involved compiler modules: ALL
- Comments:
  - ★ Keep into account the different levels of precedence (e.g., * has higher precedence than +)
- Tasks:
  - ★ Implement the following Java operators:
    - Division operator '/'
    - Remainder operator '%'
    - Shift operators '<<', '>>' (arithmetic left and right), and '>>>' (logical right)
    - Bitwise operators: '∼' (unary complement), '|' (inclusive OR), '^' (exclusive OR), '&' (AND)
    - Unary plus operator '+' (e.g., +5).
  - ★ Update accordingly j–/lexicalgrammar and j–/grammar

Step 1: Scanner

- Description: Add more Java tokens to the *j--* language
- Involved compiler modules: Scanner
- Comments:
  - ★ You only have to add **scanning** support. Hence, the only Java files to be modified in j–/src/jminusminus are TokenInfo.java and Scanner.java.
  - ★ In order to **only scan** a *j--* file, you need to run *j--* with option '**-t**'. This will scan the input *j--* file, and print the tokens in the program.
- Tasks:
  - ★ Check if the following Java tokens are supported, and add scanning support for them if not:
    - ▶ Multi-line comments: ignore all text from '/*' to '*/'
    - ▶ Reserved words:

abstract	const	finally	int	public	this
boolean	continue	float	interface	return	throw
break	default	for	long	short	throws
byte	do	goto	native	static	transient
case	double	if	new	strictfp	try
catch	else	implements	package	super	void
char	extends	import	private	switch	volatile
class	final	instanceof	protected	synchronized	while

    - ▶ Operators:

?	=	==	!	∼	!=	/	/=	+	+=	++	-
-=	–	*	*=	%	%=	>>	>>=	>>>	>>>=	>=	>
<<	<<=	<=	<	^	^=	|	|=	||	&	&=	&&

    - ▶ Separators:

,	.	{	[	(	)	]	}	;	:

  - ★ Update accordingly j–/lexicalgrammar and j–/grammar

Step 2: Parser

- Description: Add more Java constructs to the *j--* language.
- Involved compiler modules: Parser
- Comments:
  - ★ Add support for the **parsing** of some programming constructs, and for their representation in the abstract syntax tree (**AST**).
  - ★ In order to **only parse** a *j--* file, you need to run *j--* with option '-**p**'. This will print the AST of the program in XML format.
- Tasks: Add parsing support for
  - ★ double basic data type
  - ★ Logical OR operator `||`, assignment operators -=. *=, /=, %=, the prefix operator --, the postfix operator ++
  - ★ Static and instance initialization blocks (https://docs.oracle.com/javase/tutorial/java/javaOO/initial.html)
  - ★ Interface declaration
  - ★ Conditional expressions (cond ? thenBranch : elseBranch)
  - ★ The two kinds of for statements: `for(int i=1; i<n; i++)` and `for(int item : numbers)`
  - ★ Exception handling: which involves supporting `try`, `catch`, `finally`, `throw`, and `throws`

Step 3: Scanning and Parsing with JavaCC

- Description: Add more tokens and Java constructs to the *j*-- language using JavaCC.
- Involved compiler modules: Scanner and Parser
- Comments:
  - ★ Modify the JavaCC specification file *j*--/src/jminusminus/*j*--.jj
    1. First of all it is necessary to modify the scanner section of *j*--.jj to support the Java tokens from Step 1.
    2. Then, modify the parser section to support the Java programming constructs from Step 2.
- Tasks:
  - ★ Considering the tokens from Step 1, using JavaCC add scanner support for:
    - Multi-line comments, Reserved words, Operators, Separators.
  - ★ Considering the constructs from Step 2, using JavaCC add parsing support for:
    - double basic data type
    - Logical OR operator ||, assignment operators -=. *=, /=, %=, the prefix operator –, the postfix operator ++
    - Static and instance initialization blocks
    - Interface declaration
    - Conditional expressions (cond ? thenBranch : elseBranch)
    - The two kinds of for statements: `for(int i=1; i<n; i++)` and `for(int item : numbers)`
    - Exception handling, which involves supporting `try`, `catch`, `finally`, `throw`, and `throws`

Step 4: Type Checking

- Description: Implement type checking for the Java programming constructs introduced in Step 2.
- Involved compiler modules: Semantics/Type checker
- Comments:
  - ★ In order to **only pre-analyze** a *j--* file, you need to run *j--* with option '**-pa**'. This will print the AST of the program enriched by the pre-analysis phase.
  - ★ In order to **only analyze** a *j--* file, you need to run *j--* with option '**-a**'. This will print the AST of the program enriched by the pre-analysis and analysis phase
- Tasks: Considering the constructs from Step 2, add support for:
  - ★ double basic data type
  - ★ Logical OR operator `||`, assignment operators -=. *=, /=, %=, the prefix operator –, the postfix operator ++
  - ★ Static and instance initialization blocks
  - ★ Interface declaration
  - ★ Conditional expressions (cond ? thenBranch : elseBranch)
  - ★ The two kinds of for statements: `for(int i=1; i<n; i++)` and `for(int item : numbers)`
  - ★ Exception handling, which involves supporting `try`, `catch`, `finally`, `throw`, and `throws`

Step 5: JVM Code Generation

- Description: Implement JVM code generation for the Java programming constructs introduced in Step 2.
- Involved compiler modules: CodeGen
- Tasks: Considering the constructs from Step 2, add support for:
  - ★ double basic data type
  - ★ Logical OR operator ||, assignment operators -=. *=, /=, %=, the prefix operator –, the postfix operator ++
  - ★ Static and instance initialization blocks
  - ★ Interface declaration
  - ★ Conditional expressions (cond ? thenBranch : elseBranch)
  - ★ The two kinds of for statements: `for(int i=1; i<n; i++)` and `for(int item : numbers)`
  - ★ Exception handling, which involves supporting `try`, `catch`, `finally`, `throw`, and `throws`

Questions/Feedback?

Questions/Feedback?

Problems?

- Post your question on Piazza
- Ask me or Emad either via e-mail or in class

Optional challenge

"It will take us 1 hour to do this project . . . "

- Consider further Java features from Java 8 or Java 9
  - ★ Interfaces in Java 7, Java 8, and Java 9
    http://technikes.com/what-is-the-difference-in-interface-of-java7-java8-and-java9/

    Java 7: Constants, abstract methods
    Java 8: Constants, abstract methods, default methods, static methods
    Java 9: Constants, abstract methods, default methods, static methods, private methods, private static methods
  - ★ Further novelties in Java 8
    https://o7planning.org/en/10323/syntax-and-new-features-in-java-8
    - Lambda expressions
    - Reference methods
  - ★ Further features?

Analysis: How would you plan to add support for some of these features?
Can they be supported at all in the current structure of *j*--?

Develop: Implement some of these features

Tentative Lecture Plan

#	*Date*	*Topic*	*Chapter*	*Suggested project plan*
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		Complete Steps 0 and 1
6	Mar 8	Parsing	3	
7	Mar 15	Parsing 2	3	Begin Steps 2 and 3
8	Mar 22	Type Checking	4	
	Mar 29	*Easter holiday*		
9	Apr 5	Type Checking 2	4	
10	Apr 12	Lab Day		Complete Steps 2, 3, and 4
11	Apr 19	JVM Code Generation	5	Begin Step 5
12	Apr 26	MIPS Code Generation	6	
13	May 3	Register Allocation	7	
	?	Translating functional languages		
	May 10	Deadline for final assignment		Project and report ready
	May 18	Exam		

Alternative Project

"We already did something similar in previous courses . . ."

- Description: Implement a *j--* compiler using XTEXT (www.eclipse.org/Xtext/)
- Involved compiler modules: ALL
- Comments:
  - ★ XTEXT is similar to JavaCC, but has a different 'input language'
  - ★ XTEXT creates parser, scanner, as well as a modern IDE for your language.
  - ★ The project does not involve
    1. Modifying hand-written scanner and parser
    2. Extending *j--* language
- Tasks:
  - ★ Specify *j--* grammar in XTEXT (and generate the scanner and parser)
  - ★ Implement modules for analysis and codegen
    - ▶ You can re-implement such modules
      Reusing as much code as you can, e.g., CLEmitter
      Getting inspiration from *j--* source-code
    - ▶ Or you can invoke *j--* to perform the actual compilation
      Plan B: Cannot be a '12 points' project