Sharif University of Technology

Compiler Course Project

Spring 1395

Chapter 1

Project Introduction

For your course in this semester, you have to work on a compiler for a programming language called *Decaf* which will be defined in the rest of this handout. In this project you will create three parts of a compiler: scanner, parser and code generator. In order to simplify the project we will use a virtual machine to execute your generated codes. You will be provided with the virtual machine to test your compiler.

I think this is enough for the introduction. I hope you all enjoy this project and learn something useful from it. If you have any questions which you think can't be asked in the group, feel free to contact me via email.

Chapter 2

Project Definition

For the project of this course, you should write a compiler for a fictional language called *Decaf*. Language specification is in section 2.1.

2.1 Language Specification

Decaf is pretty much like Java that you know. We follow on to explain the details of this language.

2.1.1 Lexical Considerations

Keywords and identifiers in Decaf are case-sensitive. All the keywords in Decaf are lowercase and they are listed below:

```
boolean, break, char, continue, else, false, float, for, if, int, readchar,
readfloat, readint, return, true, void, while, writechar, writefloat, writeint
```

Comments are started by `//` and are terminated by the end of the line or can be of the form "`/*` *Comment* `*/`" which may include multiple lines.

White space may appear between any two consecutive lexical tokens. White space is defined as one or more spaces, tabs and line-break characters, and comments.

Integer numbers in Decaf are 32 bit signed. That is, decimal values between -2147483648 and 2147483647. If a sequence begins with 0x, then these first two characters and the longest sequence of characters drawn from [0-9][a-f][A-F] form a hexadecimal integer literal. If a sequence begins with a decimal digit (but not 0x), then the longest prefix of decimal digits forms a decimal integer literal. Note that range checking is performed later. A long sequence of digits (e.g. 123456789123456789) is still scanned as a single token. You should do the same for floating point numbers.

A ⟨`float`⟩ denotes a floating point number that is described in the grammer(p. 5).

A ⟨`char`⟩ is any printable ASCII character (ASCII values between decimal value 32 and 126, or octal

40 and 176) other than single quote (’), or backslash (\), plus the 2-character sequences \’ to denote single quote, \\ to denote backslash, \t to denote a literal tab and \n to denote newline.

2.1.2 Language Grammer

Meta-notation:

⟨foo⟩	means foo is a nonterminal.
foo	(in **bold** font) means that **foo** is a terminal i.e., a token or a part of a token.
$x^?$	means zero or one occurence of x, i.e., x is optional;
x^*	means zero or more occurrences of x.
$\left\{ \quad \right\}$	large braces are used for grouping; note that braces in quotes ′{′ ′}′ are terminals.
|	separates alternatives.

⟨program⟩ → $\left\{$ ⟨var_dec⟩ | ⟨method_dec⟩ $\right\}^*$

⟨var_dec⟩ → ⟨type⟩ ⟨var_list⟩ ;

⟨var_list⟩ → ⟨id⟩ $\left\{$ [⟨int_literal⟩] $\right\}^*$ $\left\{$, ⟨var_list⟩ $\right\}^?$

⟨method_dec⟩ → ⟨ret_type⟩ ⟨id⟩ (⟨method_list⟩$^?$) ⟨block⟩

⟨method_list⟩ → ⟨type⟩ ⟨id⟩ $\left\{$, ⟨method_list⟩ $\right\}^?$

⟨block⟩ → ′{′ $\left\{$ ⟨var_dec⟩ | ⟨statement⟩ $\right\}^*$ ′}′

⟨type⟩ → **int** | **boolean** | **float** | **char**

⟨ret_type⟩ → **void** | ⟨type⟩

⟨statement⟩ →
- ⟨assignment⟩ ;
- ⟨method_call⟩ ;
- **if (** ⟨expr⟩ **)** ⟨block⟩ $\left\{$**else** ⟨block⟩$\right\}^?$
- **while (** ⟨expr⟩ **)** ⟨block⟩
- **for (** ⟨assignment⟩ ; ⟨expr⟩ ; ⟨assignment⟩ **)** ⟨block⟩
- **return** ⟨expr⟩$^?$;
- **break** ;
- **continue** ;
- ⟨block⟩
- $\left\{$ **readfloat** | **readint** | **readchar** $\right\}$ ⟨location⟩ ;
- $\left\{$ **writefloat** | **writeint** | **writechar** $\right\}$ ⟨expr⟩ ;

$$
\begin{array}{rcl}
\langle \texttt{assignment} \rangle & \rightarrow & \langle \texttt{location} \rangle = \langle \texttt{expr} \rangle \\
\langle \texttt{method\_call} \rangle & \rightarrow & \langle \texttt{method\_name} \rangle \ \texttt{(} \ \langle \texttt{parameter\_list} \rangle^{?} \texttt{)} \\
\langle \texttt{parameter\_list} \rangle & \rightarrow & \langle \texttt{expr} \rangle \ \left\{ \ \texttt{,} \ \langle \texttt{parameter\_list} \rangle \ \right\}^{?} \\
\langle \texttt{method\_name} \rangle & \rightarrow & \langle \texttt{id} \rangle \\
\langle \texttt{location} \rangle & \rightarrow & \langle \texttt{id} \rangle \left\{ \ \texttt{[} \ \langle \texttt{expr} \rangle \ \texttt{]} \ \right\}^{*} \\
\langle \texttt{expr} \rangle & \rightarrow & \langle \texttt{location} \rangle \\
 & | & \langle \texttt{method\_call} \rangle \\
 & | & \langle \texttt{literal} \rangle \\
 & | & \langle \texttt{expr} \rangle \ \langle \texttt{bin\_op} \rangle \ \langle \texttt{expr} \rangle \\
 & | & -\langle \texttt{expr} \rangle \\
 & | & !\langle \texttt{expr} \rangle \\
 & | & \texttt{(} \ \langle \texttt{expr} \rangle \ \texttt{)} \\
\langle \texttt{bin\_op} \rangle & \rightarrow & \langle \texttt{arith\_op} \rangle \ | \ \langle \texttt{rel\_op} \rangle \ | \ \langle \texttt{eq\_op} \rangle \ | \ \langle \texttt{cond\_op} \rangle \\
\langle \texttt{arith\_op} \rangle & \rightarrow & \texttt{+} \ | \ \texttt{-} \ | \ \texttt{*} \ | \ \texttt{/} \ | \ \texttt{\%} \\
\langle \texttt{rel\_op} \rangle & \rightarrow & \texttt{<} \ | \ \texttt{>} \ | \ \texttt{<=} \ | \ \texttt{>=} \\
\langle \texttt{eq\_op} \rangle & \rightarrow & \texttt{==} \ | \ \texttt{!=} \\
\langle \texttt{cond\_op} \rangle & \rightarrow & \texttt{\&\&} \ | \ \texttt{||} \\
\langle \texttt{literal} \rangle & \rightarrow & \langle \texttt{int\_literal} \rangle \ | \ \langle \texttt{bool\_literal} \rangle \ | \ \langle \texttt{float\_literal} \rangle \ | \ \langle \texttt{char\_literal} \rangle \\
\langle \texttt{id} \rangle & \rightarrow & \langle \texttt{alpha} \rangle \ \langle \texttt{alpha\_num} \rangle^{*} \\
\langle \texttt{alpha\_num} \rangle & \rightarrow & \langle \texttt{alpha} \rangle \ | \ \langle \texttt{digit} \rangle \\
\langle \texttt{alpha} \rangle & \rightarrow & \texttt{a} \ | \ \texttt{b} \ | \ \ldots \ | \ \texttt{z} \ | \ \texttt{A} \ | \ \texttt{B} \ | \ \ldots \ | \ \texttt{Z} \ | \ \texttt{\_} \\
\langle \texttt{digit} \rangle & \rightarrow & \texttt{0} \ | \ \texttt{1} \ | \ \texttt{2} \ | \ \ldots \ | \ \texttt{9} \\
\langle \texttt{hex\_digit} \rangle & \rightarrow & \langle \texttt{digit} \rangle \ | \ \texttt{a} \ | \ \texttt{b} \ | \ \texttt{c} \ | \ \texttt{d} \ | \ \texttt{e} \ | \ \texttt{f} \ | \ \texttt{A} \ | \ \texttt{B} \ | \ \texttt{C} \ | \ \texttt{D} \ | \ \texttt{E} \ | \ \texttt{F} \\
\langle \texttt{int\_literal} \rangle & \rightarrow & \langle \texttt{decimal\_literal} \rangle \ | \ \langle \texttt{hex\_literal} \rangle \\
\langle \texttt{decimal\_literal} \rangle & \rightarrow & \langle \texttt{digit} \rangle \ \langle \texttt{digit} \rangle^{*} \\
\langle \texttt{hex\_literal} \rangle & \rightarrow & \texttt{0x} \ \langle \texttt{hex\_digit} \rangle \ \langle \texttt{hex\_digit} \rangle^{*} \\
\langle \texttt{bool\_literal} \rangle & \rightarrow & \textbf{true} \ | \ \textbf{false}
\end{array}
$$

$$
\begin{array}{rcl}
\langle \texttt{float\_literal} \rangle & \rightarrow & \langle \texttt{decimal\_literal} \rangle \, . \, \langle \texttt{decimal\_literal} \rangle \\
& | & . \, \langle \texttt{decimal\_literal} \rangle \\
& | & \langle \texttt{decimal\_literal} \rangle \, . \\
\\
\langle \texttt{char\_literal} \rangle & \rightarrow & \texttt{'} \, \langle \texttt{char} \rangle \, \texttt{'}
\end{array}
$$

2.1.3 Semantic

A program in Decaf consists of zero or more variable declaration and at least one function declaration. The program must contain a declaration for a method called **main** that has no parameters and has return type **void** . Execution of a program starts at method **main** .

2.1.4 Types

There are four basic types in Decaf — **int** , **boolean** , **float** and **char** . In addition, there are arrays of these types ($\langle \texttt{type} \rangle [N_1] \dots [N_n]$).

All arrays should have compile-time fixed size and are indexed from 0 to $N_i - 1$, where $N_i > 0$ is the size of the array.

2.1.5 Scope Rules

Decaf has simple rules on its scopes:

- a variable must be declared before it is used.
- a method can be called only by code appearing after its header.

There are multiple valid scopes at any point of a program. When the code flow enters a new block, a new scope is created and variables declared in this scope will shadow variables in previous scopes (in the case having same names). And like languages similar to C when code flow leaves a block, the related scope and all variables declared in it will vanish and can't be used after that.

Variable names defined in the method scope may shadow method names in the global scope. No identifier may be defined more than once in the same scope. Thus field and method names must all be distinct in the global scope, and local variable names and formal parameter names must be distinct in the first scope of each method.

2.1.6 Location

There is no default value for locations when they are delared. So there is no need to initialize the variables when we reach their declaration.

2.1.7 Assignments

Assigning two variables means copying the contents of ⟨`expr`⟩ to ⟨`location`⟩, so unlike C you can copy an array by a single assignment! Note that the ⟨`location`⟩ and the ⟨`expr`⟩ in an assignment must have the same type.

(It's not possible to copy an ⟨`int`⟩ into a ⟨`float`⟩ and vice versa)

2.1.8 Method Invocation and Return

Argument passing is defined in terms of assignment: the formal arguments of a method are considered to be like local variables of the method and are initialized, by assignment, to the values resulting from the evaluation of the argument expressions. The arguments are evaluated from left to right.

A method that returns a result may be called as part of an expression, in which case the result of the call is the result of evaluating the expression in the return statement when this statement is reached. It is illegal for control to reach the textual end of a method that returns a result. A method that returns a result may also be called as a statement. In this case, the result is ignored.

A method that has no declared result type can only be called as a statement, i.e., it cannot be used in an expression. Such a method returns control to the caller when return is called (no result expression is allowed) or when the textual end of the callee is reached.

2.1.9 Expressions

Expressions follow the normal rules for evaluation. In the absence of other constraints, operators with the same precedence are evaluated from left to right. Parentheses may be used to override normal precedence.

Operator precedence, from highest to lowest:

Operators	*Comments*
`-`	unary minus
`!`	logical not
`* / %`	multiplication, division, remainder
`+ -`	addition, subtraction
`< <= >= >`	relational
`== !=`	equality
`&&`	conditional and
`||`	conditional or

Note that this precedence is not reflected in the reference grammar.

2.1.10 Semantic Rules

These rules place additional constraints on the set of valid *Decaf* programs besides the constraints implied by the grammar. A program that is grammatically well-formed and does not violate any of

the following rules is called a legal program. A robust compiler will explicitly check each of these rules, and will generate an error message describing each violation it is able to find. A robust compiler will generate at least one error message for each illegal program, but will generate no errors for a legal program.

1. No identifier is declared twice in the same scope.
2. No identifier is used before it is declared.
3. The program contains a definition for a method called **main** that has no parameters (note that since execution starts at method **main** , any methods defined after main will never be executed).
4. The ⟨`int_literal`⟩ in an array declaration must be greater than 0.
5. The number and types of arguments in a method call must be the same as the number and types of the formals i.e., the signatures must be identical.
6. If a method call is used as an expression, the method must return a result.
7. A **return** statement must not have a return value unless it appears in the body of a method that is declared to return a value.
8. The expression in a return statement must have the same type as the declared result type of the enclosing method definition.
9. An ⟨`id`⟩ used as a ⟨`location`⟩ must name a declared local/global variable or formal parameter.
10. For all locations of the form ⟨`id`⟩[⟨`expr`⟩]...[⟨`expr`⟩]
    (a) ⟨`id`⟩ must be an **array** variable, and
    (b) the type of ⟨`expr`⟩ must be **int** .
11. The ⟨`expr`⟩ in **if** statement must have type **boolean** .
12. The operands of ⟨`arith_op`⟩s must have type **int** or **float**.
13. The operands of ⟨`rel_op`⟩s must have type **int** , **float** or **char**.
14. The operands of ⟨`eq_op`⟩s must have the same type , either **int** , **float** , **boolean** or **char** .
15. The operands of ⟨`cond_op`⟩s and the operand of logical not (`!`) must have type **boolean** .
16. The ⟨`location`⟩ and the ⟨`expr`⟩ in an assignment, ⟨`location`⟩ = ⟨`expr`⟩ must have the same type.
17. The ⟨`expr`⟩ of **for** must have type **boolean** .
18. All **break** and **continue** statements must be contained within the body of a **for** or a **while** .

2.1.11 Run Time Checking

In addition to the constraints described above, which are statically enforced by the compilers semantic checker, the following constraints are enforced dynamically: the compilers code generator must emit code to perform these checks; violations are discovered at run-time.

1. The subscript of an array must be in bounds.
2. Control must not fall off the end of a method that is declared to return a result.

Chapter 3

Scanner

You have found out enough about Decaf in the previous handouts, now it's time to start the project. After defining our preferred language we can take the first step and that would be preparing a scanner to tokenize any programs written using it.

3.1 Functional Difference

As you have learned in the class *(I hope so)*, scanner's job is to parse a program and pass found tokens to parser for further processes. For this part you have to do the same things but with some slight changes.

As roughly mentioned before, our compiler should not accept invalid literals in a program but scanner is not fully able to do it. An integer which is not fit in a 32-bit signed format is an example of these invalid literals. Now consider -2147483648 which is a valid 32-bit signed integer. An scanner which faces these characters in the input will detect two tokens, "`-`" and "`int_literal`". But this "`int_literal`" is not a 32-bit signed integer so scanner can't tell if this is a correct "`int_literal`" or not because it doesn't know the kind of "`-`" (unary or binary). To solve this problem, after detecting the token type of a literal, instead of using `ICV` which is an integer type variable, scanner will use an string variable and leave further processes for other parts. To make it general, do the same for other kinds of literals *(Don't worry, this will make implementation a little easier)*. For this purpose, use a public variable (string typed) in your scanner named `CV` (means Constant Value).

3.2 What to pass

Scanner should pass tokens that finds in input to parser by the means of "`NextToken`" method. In order to make it unique between all the projects, we will define them here.

The scanner should return an string which determines the scanned token. For variables return "`id`" (note that the variable names will be stored in `CV` in order to be available for other parts of the compiler), for every literal use its type concatenated with "`_literal`" ("`int_literal`", "`float_literal`", "`bool_literal`" and "`char_literal`".) and for all other tokens, return their form that appears in the language grammar.

You can see an input program and the scanner's result for it.

"`if ( i >= 0 > = 12 )`" → "`if ( id >= int_literal > = int_literal )`"

Parser will receive one of these tokens for each call of "`NextToken`" (first "`if`", second "`(`", third "`id`" and so on) .

3.3 Implementation

Your implementation for this phase should contain at least a class named "`Scanner`" containing (at least) a constructor and a public method. The constructor will have only one parameter, a string containing the filename (and address) of the source code that your scanner should tokenize and the method (which should be named "`NextToken`") will have no parameter and will return a string. If you are using Java, use "`Compiler`" as your namespace/package name.

If your scanner faces an error in the input file, it must throw an exception with a message containing the error description. If the token in input is not recognized, the message should be "`Invalid Token at line XXX = Unknown_Token`". This means your scanner should be able to recognize the line number when tokenizing the input. For example:
`Invalid Token at line 1 = @`

Hint: *As you will notice later, you need to write reserved words somewhere in your code to make difference between them and "id"s.*

3.4 Extra bonus grade

Supporting `\#includes` is the scanner's job. Implementing this feature has an extra grade of 0.3. Note that each file included contains a ⟨`program`⟩. Merging all the ⟨`program`⟩s into one is your job.

3.5 Extra notes

When implementing the scanner you AREN'T ALLOWED to use the built-in regular expressions.

Chapter 4

Parser

After obtaining a scanner for our compiler, we need a parser. The preferred way to do so in this project is implementing a Recursive Descent Parser. As you have studied in the course, a recursive descent parser uses a Top-Down approach to parse a tokenized program.

For more information about recursive descent parsers, and some examples refer to this slide.

4.1 Hacked Grammar

The grammar provided in 2.1 just presents the Decaf's language but is not suitable for a top-down parser since left recursions are possible in the grammar. Additionally, the operator precedence is not included in that grammar.

In order to make the grammar compatible with your parser, you have to construct a LL(1) grammar based on the original grammar. (You may find Left Factoring technique useful) Be careful not to change the language.

4.2 Implementation

Implementing a recursive descent parser doesn't need an external representation of the grammar. In this method, the goal is providing a set of mutually recursive functions which represent the desired language. Each of this functions denotes one of non-terminal nodes in the grammar and the overall structure of these functions should matches with the structure of the grammar.

Your implementation for this phase should contain at least a class named "`Parser`" containing (at least) a constructor and a public method. The constructor will be same as the scanner's and will have only one parameter, filename of source code, and the method (which should be named "`Parse`") will have no parameter, will return nothing and parses the source code by calling the function denoting the ⟨`program`⟩ non-terminal. If you are using Java, use "`Compiler`" as your namespace/package name.

If one of your functions wasn't able to match the current token and returned **false**, your parser must throw an exception with a message containing the unexpected token, formatted like "`Unexpected Token at line XXX = Unexpected_Token`".

As an example suppose this is the source code:

```
void 45(int a) { }
```

The corresponding error should be:

```
Unexpected Token at line 1 = int_literal
```

4.3 Note

Any other method, such as implementing a shift-reduce parser, is also acceptable as long as it works and is a one-pass method, but using a parser generator is prohibited. You may want to choose a different method at your own risk, but don't expect any help from us when you encounter a serious problem during your project.

Chapter 5

Code Generator

After preparing scanner and parser, we can continue with code generator *(I won't explain anymore about CG and will continue with implementation details).*

5.1 Virtual Machine

Virtual machine works after the compiler has done its job, and executes the output of the compiler. The virtual machine and the compiler need not to be seperate programs.

5.2 Implementation

The final result of this project is an executable file, say compiler-vm.jar (or compiler-vm.exe). The executable takes two arguments, the first of which is the source file we want to compile and the other is the file the compiled source is written to, and compiles the source code. After the source is compiled, the virtual machine runs the program. So executing the following line should compile and run "`program.L`" and also write the compiled source to "`output.Lm`".

```
java -jar compiler-vm.jar program.L output.Lm
```

5.3 Errors

You will report any errors you see during the compiling process in form of exceptions including semantic errors (just like what you did in scanner and parser).

5.4 Extra Bonus Grade

If you optimize the generated code, you get up to 1 grade (It depends on the level of optimization).

Chapter 6

Virtual Machine

Instructions appear in one line of output each and will have the following format:

[OpCode] [Opr_1] ... [Opr_n]

Where the number of operands (n) is determined for each instruction separately. All of the required instructions in *Decaf* are listed in table below. If you need any instructions that is not listed here, please inform me.

Operator Name	OpCode	# of Operands	Operation
Add	+	3	[Opr1] ← [Opr2] + [Opr3]
Subtract	−	3	[Opr1] ← [Opr2] − [Opr3]
Multiply	*	3	[Opr1] ← [Opr2] * [Opr3]
Divide	/	3	[Opr1] ← [Opr2] / [Opr3]
Mod	%	3	[Opr1] ← [Opr2] % [Opr3]
Logical And	&&	3	[Opr1] ← [Opr2] && [Opr3]
Logical Or	||	3	[Opr1] ← [Opr2] || [Opr3]
Less Than	<	3	[Opr1] ← [Opr2] < [Opr3]
Greater Than	>	3	[Opr1] ← [Opr2] > [Opr3]
Less Than Equal	<=	3	[Opr1] ← [Opr2] <= [Opr3]
Greater Than Equal	>=	3	[Opr1] ← [Opr2] >= [Opr3]
Equal	==	3	[Opr1] ← [Opr2] == [Opr3]
Not Equal	!=	3	[Opr1] ← [Opr2] != [Opr3]
Logical Not	!	2	[Opr1] ← ! [Opr2]
Assignment	:=	2	[Opr1] ← [Opr2]
Jump If True	*jt*	2	if [Opr1]==TRUE then pc ← [Opr2]
Jump	*jmp*	1	pc ← [Opr1]
Write Integer	*wi*	1	{output} ← [Opr1]
Write Float	*wf*	1	{output} ← [Opr1]
Write Boolean	*wb*	1	{output} ← [Opr1]
Write Character	*wc*	1	{output} ← [Opr1]
Read Integer	*ri*	1	{input} →[Opr1]
Read Float	*rf*	1	{input} →[Opr1]
Read Boolean	*rb*	1	{input} →[Opr1]
Read Character	*rc*	1	{input} →[Opr1]

Operator Name	OpCode	# of Operands	Operation
PC Value	:= *pc*	1	[Opr1] ← pc
SP Value	:= *sp*	1	[Opr1] ← sp
Assign SP	*sp* :=	1	sp ← [Opr1]
Return	*ret*	2	pc ← [Opr1] , sp ← [Opr2]

6.1 Operands

Each of the operands will be of the following format:

[Addressing Mode] [Type] [Value]

You have to concatenate their text values in order to obtain the operand. For immediate addressing, value will be the literal except characters. The immediate value for characters should be their ASCII value. In other kinds of addressing, value will be a memory address (integer).

6.2 Addressing Modes

in *Decaf* we will need at most five kind of addressing mode.

Addressing Mode	Text Form
Global Direct	gd_
Global Indirect	gi_
Local Direct	ld_
Local Indirect	li_
Immediate	im_

6.3 Types

Type	Text Form
Integer	i_
Float	f_
Boolean	b_
Char	c_

6.4 Example

In this section you can see some examples from instructions in text form. the white space between operator and operands can be a single space or tab.

```
+   gd_i_12  im_i_5     ld_i_14
wc  im_c_13
*   gi_f_10  im_f_10.5  ld_f_10
```