Practical Sessions

Aims

The aim of the practical sessions is to give you practical experience of Formal Language Processing techniques. It isn't easy to write a compiler so we will start from existing software and make changes to it.

René Descartes wrote:

"I should like the reader first of all to go quickly through the whole book like a novel, without straining his attention too much or stopping at the difficulties which may be encountered. The aim should be merely to ascertain in a general way the matters I have dealt with. After this, if he finds that these matters deserve to be examined and he has the curiosity to ascertain their causes, he may read the book a second time in order to observe how my arguments follow. But if he is not able to see this fully, or if he does not understand all the arguments, he should not give up at once. He should merely mark with a pen the places where he finds the difficulties and continue to read on to the end without a break. If he then takes up the book for a third time, I venture to think he will find the solutions to most of the difficulties he marked before; and if any still remain, he will discover their solution on a final re-reading."

This is also very good advice when looking at software. The software I'm providing consists of about 20 Java classes which implement a compiler. Some of them you will probably never look at. There are about eight you will learn something about.

For the first sessions I will make things very easy: this is a bit like Descartes' first reading – you'll ignore 90% of what's in the program and just make a few small changes. For the first practical, I'll tell you exactly what you need to do, which files to look in and what to change.

Later I'll give you more freedom to explore and make some bigger changes. Like Descartes' second stage of reading you will have to look more carefully at the code and, I hope, will understand more of it. We'll answer any questions you have so you don't have to "mark...the difficulties".

The assessment of the course will require you (working in groups) to make some larger changes to the code. The details are at the end of this document.

Getting started

The compiler is available as source from dropbox as SMALL.zip. Download and unzip to a suitable place. This will give you a folder called SMALL which contains sub-directories:

SMALL

w *small Windows/DOS batch files to build and run the compiler*

clean.bat	*delete class files*
build.bat	*recompile changed java sources*
small.bat	*compile a SMALL program*
run.bat	*run a compiled SMALL program*
smallgo.bat	*compile and run*
jasmin.jar	*the 'assembler' - ignore*
small.jar	*the compiler - ignore for now*

b *small bash shell scripts to build and run the compiler*
as for w but no '.bat' file extension

examples *small example programs*

sal *the main software*

Library.java	*SMALL's run time library*

small ***classes specific to SMALL***

CodeGen.java	*convert AST into Jasmin code*
Code.java	*low level code output*
Scope.java	*managing 'blocks' and names*
Descriptor.java	*types of thing stored in scope*
Main.java	*start the parser*
Parse.java	*read SMALL program, produce AST*
Token.java	***enum** of tokens for Scanner to use*
Tree.java	*used by Parse to build AST*.

util ***utilities***

CharView.java	*manipulate of parts of strings*
ErrorStream.java	*error messages*
Lexer.java	*general scanner*
Patterned.java	*interface used by Scanner*
RE.java	*regex simplifier*
Templater.java	*simple template engine*
Fail.java	*detecting logical errors*

You can compile everything at a terminal by making flp the current folder and typing:

w\build.bat (Windows) or ./b/build (Linux)

There will be a warning:

Note: Some input files use unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

This can be ignored (other error messages can't!).

After using build.bat a SMALL program can be compiled by typing:

```
w\small  HelloWorld.sm          (linux) ./b/small  HelloWorld.sm
w\run    HelloWorld             (linux) ./b/run    HelloWorld
```

The small command assume a full file name, any extension (or none) is allowed. `run` assume the name without extension. Wherever the original file is, the output is always in the current directory. So (I'll just use windows example from now on):

```
w\small examples\HelloWorld.sm
```

produces HelloWorld.j (an assembler text file) and HelloWorld.class (a java class file) in your current directory.

The command

```
w\smallgo examples\HelloWorld
```

compiles and runs a program assumed to have an extension .sm (the folder `examples` contains several simple programs).

Netbeans users can create a project from the files which some may find easier. If you do so you will initially get an error message from Netbeans referring to `jasmin.Main.main` this can be removed by going to project settings and adding the jasmin.jar file to the list of libraries.

0 Getting To Know the Compiler

0.1 Build the compiler

Down load the files, build the compiler. Look at the examples and make sure you understand enough of SMALL to know what they are doing. Check that you can compile and run the examples.

0.2 Write a SMALL program

Write a SMALL program to read in three numbers, a, b, c and print out something like "1 + 2*3 = 7" (assuming a = 1, b=2, c=3). Check the program works and look at the .j file which was produced as well as .class file. Can you understand what the various instructions do?

Repeat this for a program containing a simple loop or a if-statement. You don't have to become an expert on JVM but should have general sense of what is happening.

1 The Lexer and Regular Expressions (mostly)

1.0 Learning about Regular Expressions

If you aren't familiar with regular expressions in Java (or other languages) I'd advise learning about them. They can be very useful for any data validation problem.

The Java Regular Expressions are described here. There are many extra features beyond those mentioned in class. The java string class also allows the use of regular expressions, e.g. "1234, 223,, aaa".replaceAll(",\\w+,|[0-9]+", "0"); Can you predict the result of this operation?

My Java module for making Regular Expressions more readable is in sal.util.RE.java. A summary of it can be found in the **docs** folder of **SMALL**. Another library which uses English language like REs is Simple-Regex.

Regular Expressions 101 allows you enter Regular Expressions, see what they match, get an explanation of what's happening and can even generate language specific Expressions you can paste into your programs.

As a test, try the following. Java now allows _ characters in numbers to make them more readable. For example you can write 1_234_456 Can you describe this with an RE? Test to see if it works. Harder: don't allow numbers to start or end with _ however 1___2 is allowed.

1.1 Create a French Programming Language

Look at the file Token.java. Token is a Java **`enum`** type which means it has a fixed number of objects, all of which are defined in the file. Ignore the details of what the arguments to the tokens mean – the names (in capital letters, eg IF, THEN, END, EQ etc) are what is important.

Some of the Tokens have as arguments simple strings, e.g THEN("then") means that the token THEN corresponds to seeing **`then`** in a SMALL program, i.e. it is a reserved or key-word. Leave the Token names (IF, THEN, END …) unchanged but replace the texts ("if", "then", "end"). with the French equivalents (avoid accents – I haven't tested that!) and use BUILD.BAT to compile the SMALL compiler. You now have a French version of the language. Modify one of the example programs to have French keywords and check it still works.

This is a standard feature of compilers: the lexical analysis deals with text, the rest of the program doesn't care how reserved words are written.

You can continue to use the French version if you wish.

1.2 Implement break/continue

Some of the tokens defined in Token have no text pattern associated with them. For example near the beginning you will find:

```
WHILE("while"), DO("do"), UNTIL,
```

This informs the tokeniser that the WHILE token should be returned when it finds the word 'while' in its input, the DO token when it sees 'do' but what about UNTIL? Since it has no text, it will never be returned by the tokeniser. The code that uses the the tokens (in Parse.java and CodeGen.java) in some case knows what to do when it sees one of these tokens – but it never does see them so that part of the code is never executed.

Similarly, SMALL doesn't have **break** and **continue** statements. There are tokens called BREAK and CONTINUE but, although the code to process them is in the compiler, SMALL treats them just as identifiers and not as keywords. What is the problem? All that is needed to correct it is that the tokens should be given appropriate text patterns so they can be recognised.

Put in the appropriate text patterns, re build the compiler and **break** and **continue** now work as key words. Write a SMALL program to check they work.

1.3 Better String Literals

In token the definition of STRING is

```
STRING(DQUOTE+any(notIn(DQUOTE))+DQUOTE, "<string>")
```

The RE is `DQUOTE+any(notIn(DQUOTE)) + DQUOTE` which is a double quote mark (") followed by any number of characters which aren't (notIn) double quotes. So you can't have *"with a " in the middle"* or even *"with a \" in the middle".* In both cases the lexer will stop too soon. Can you find a solution which allows *"with a \" in the middle"?* You should also make "\n" legal – no other escape sequences are necessary. Hint: a string is composed either of single ordinary characters or double characters, the first of which is '\'.

1.4 Hexadecimal Numbers

Find the declaration `NUMBER(some(in(DEC)), "<number>")` which defines a number in SMALL. 'some' and 'in' are methods in the class RE which tries to make regular expressions a bit simpler. `in(DEC)` is equivalent to `[0-9]` and `some(x)` means 1 or more occurrences of x. So the expression is equivalent to [0-9]+ (which you can use if you prefer). For more complicated REs it avoids the problems of remembering when to put in \\ or \\\\ and avoiding spaces.

`"<number>"`, the second argument is used for error messages like "<number> expected" and can be ignored.

This assignment will add hex numbers with formats like #1FAB. (In addition to DEC above you may find HEX useful). We could add a new token for a hex number but instead we will include it in NUMBER.

Write down an RE for a hex number. The 'easy' form of RE uses '+' to join things together so "X" + some("Y") would be equivalent to "XY+". RE also has a method oneOf(A, B, C) which is equivalent to A | B | C . Modify NUMBER so it's RE is either a hex number or a decimal one.

The parser is already set up to deal with hexadecimal numbers but because the current RE only allows decimal ones it never gets activated. Once you have modified the RE for a NUMBER rebuild the compiler. It should now allow hexadecimal numbers.

1.6 Add true / false

This assignment adds definitions of two new tokens to SMALL: TRUE and FALSE. In this case however, unlike BREAK & CONTINUE the compiler has no code to deal with them if they do appear so a small modification to part of the compiler is needed.

First, add patterns for the tokens. Inside SMALL **`false`** will be 0, **`true`** will be 1. When the compiler sees these it should convert them to the appropriate number.

Having added them to Token.java, open Parse.java (which builds the AST) and find the method `term()`. The line beginning "`case NUMBER:`" which deals with a number by adding a 'leaf' to the tree. The detail isn't important but **`value`** holds the number detected in the input (stored as a string) and **`token`** is the token type so adding an additional entry:

```
case TRUE:     t = leaf(NUMBER, "1"); break;
```

will make **true** equivalent to `1` – and similarly for **`false`** `using "0"`. Unfortunately there isn't much you can do with **`true`**/**`false`** in SMALL yet – except check that **`print true`** does what you expect and you can say **`while true do.`**

Note that the default case in term() produces an error message which lists the expected possible tokens. A perfectionist will add TRUE and FALSE to the list!

1.7 A Halt Statement

The final assignment in this section will add a new – but very simple - command to SMALL: **`halt`** causes a program to exit immediately. It will use some things you've already seen and some you haven't.

a) add a new token HALT with appropriate definition to Token.java.

This means the lexer will recognise halt as a SMALL token.

b) in Parse.java add a new case to method StatementList() for **halt**.

*This means the parser will recognise **halt** as a (very simple) statement*

Method `statementList()` near the top of Parse.java includes the lines:

```
case BREAK:
case CONTINUE:
          scan();
          aStatement = leaf(token);
```

These deal with the single word statements: **break** and **continue**. Insert a HALT case before BREAK. You have now invented a new token and added the syntax analysis necessary to include it in the AST.

c) Finally, go to CodeGen.java, near line 120 you will find the big `switch` statement which does all the code generation for statements. In this case just add a new `case`:

```
case HALT : emit("return");   break;
```

(The code produced by SMALL becomes the body of a method. The assembler command "return" exits from a method)

This exercise has shown you the main parts of a compiler even if you don't understand everything that is happening. Part a) was about adding a new token/lexeme to the language. Part b) made it a syntactical part (it is a sort of statement, and it is represented in the AST). Finally, when code generation is needed, the HALT is recognised and appropriate assembler instruction is produced.

Build the compiler again and create a modified version of Fib.sm

```
if n <= 0 then
     print "Number must be > 0\n"  halt
end
```

The rest of the program no longer needs an **`if`** statement around it.

2 Parsing

In this section will assume you can add definitions for existing tokens to Token.java. This section is about Parsing and creating an AST (Abstract Syntax Tree). The code for parsing is in Parse.java, the classes to create a Tree are in Tree.java. Changing the syntax of SMALL may require a change to the code that is generated – but most will not.

Parsing has two parts to it: recognising source input and converting it to a tree structure. Detecting tokens in the input uses several static small methods which can be found at the end of Token.java. They are used in almost every method in Parser.java so it is important to know what they do:

Token scan()	Reads the next token from the input. It also returns the token but that isn't used much.
Token currentToken()	Returns the last token read.
String currentText()	Returns the text of the last token read. If the token was, say, PLUS it will just be "+" but if it was NUMBER it will be the the number (as a string) or for STRING it while be the string read (including the quote marks).
boolean skipToken(Token... tokens)	Checks if the current token matches something in the list. If it does it will call scan() to move on to the next token. Usually called with only one argument, e.g. skipToken(ELSE).
boolean mustBe(Token... tokens)	Used when a particular token is expected. Similar to skipToken() but will automatically generate an error message if the token isn't found.

Here's an example:

```
public static Tree<Token> readStatement() {
       scan();  // skip the 'print' token        // 1
       Tree<Token> readList = list(READ);        // 2
       do {
           String name = currentText();          // 3
           mustBe(IDENTIFIER);                   // 4
           readList.addChild(leaf(IDENTIFIER, name));
                                                 // 5
       } while (skipToken(COMMA));               // 6
       return readList;                          // 7

   }
```

This describes the production of an AST for a read statement.

1. The method was called from Statement() so we know the current token must be READ or this method wouldn't have been called!
2. A tree node is really just a list of other tree nodes – hence the name list.
3. The first thing after READ should be a name, this should be the text of it.
4. Check it really is a name;
5. Add the name (a leaf node) to the READ statement's list of children.
6. A COMMA means there is another name to come.
7. Return readList, an AST which contains a list of names for data to be read into.

2.1 do/end, do/until

Extend the grammar of SMALL to include two new types of loop. Write the grammar so that only one token look ahead is needed.

Code between **do** and **end** is executed repeatedly you can only exit it by using **break** (**continue** jumps to the start of the loop). **do**/**until** introduces the equivalent to Java's do/while loop. In English **until** is the opposite of **while.**

```
do
        x = x + 1
until x == 3
```

will exit from the loop when x == 3 is true. No **end** is needed after the test.

The AST of a **while** statement consists of:

Token: WHILE

Child 0: the AST of the expression to be tested. If this is null no test is made.

Child 1: the AST of the statements in the statements in the loop.

So a **do** / **end** loop is just like a **while** but with no test. The code generation will handle it.

The AST of an **until** statement is identical except the token is UNTIL not WHILE.

Find the method doStatement() in Parse.java. Follow the instructions in it, looking at other statements to see how mustBe(...) and scan() are used.

If you use the correct format for the AST, the code generator will do the rest!

2.2 if / elif / else

Extend the if statement to allow **elif** (like `else if` in java) and **else**.

This would allow for, example,

```
if x == 1 then
     print “hello\n”
elif x == 2 then
     print “good bye\n”
else
     print “are you coming or going?\n”
end
```

The code generator expects the children to an **if** AST to be**:**

Token: IF

Child 0: test from **if** part (null for **else**)

Child 1: statement to generate if test is true

Child 2: test for first elif (if any)

Child 3: statement to generate first elif test is true

etc

If there is an **else** it should be the last pair of items in the list. **else** and **elif** are both optional.

Assignment 2.4

Java includes increment and decrement operations: ++ and -- . When a tokeniser detects ‘+’ how does it know whether it is an add operation or the first character of ‘++’? Different tokenisers have different policies. My Lexer module doesn’t specify which it chooses. The solution is *look ahead*. As an example in Token.java

```
ASSIGN("="+notBefore("="), “=”)   and   EQ("==")
```

(Ignore the second parameter to `ASSIGN` – it’s how the token appears in error messages.) `ASSIGN` will match an ‘=’ which doesn’t come before a second ‘=’ , `EQ` will match ‘==’.

Here is another example: `GT(">" + notBefore(">”,”="), ">")` this represents ‘>’ but not if the next character is ‘>’ (‘>>’ is a shift operation) or ‘=’ (‘>=’ is greater than or equal).

Code for postfix versions of the operator are already in Parse.java, once you’ve defined the tokens and rebuilt the compiler you should be able to use X++ or X--. Note that the code for recognising the postfix forms appears as part of the assignment statement code. Why? Using the code there as a guide can you add a new statement to handle the prefix version (it will do exactly the same as the postfix, but the operator comes before the name).

2.5 Shifts

This assignment requires a change to how expressions are parsed. At the moment java shift operators >>, << and >>> are defined, but Parse.java doesn't check for them.

What precedence do you think they should have? If you see, for example, 3*8 >> 1+2 does it mean (3*8) >> (1+2) or 3*(8 >> 1)+2 or something else? I think >> should (perhaps) have higher priority that *, % or /. Which of the two alternatives for in the last sentence does that mean? Look in Parse.java at the methods expression(), relopExpression(), addExpression(), multExpression() and term(). Add a new method shiftExpression() and make appropriate changes to the other methods.

2.6 Some Ambiguity

Consider this code:

```
if x < 3 then break end
```

It would be prettier if you could write:

```
if x < 3 break
```

and of course you could have:

```
if x < 3 then
     … do something
elif x > 5 continue
else break
```

I.e in **if** or **elif** you can use just **break** as an alternative to then **break end** (similarly for **continue**). Strictly this introduces an ambiguity into SMALL when used with **else**

```
else
     break
     x = 3
end
```

would still be legal. It isn't feasible to change the grammar of SMALL to make it allow for both ways of using **else** with **break** or **continue** (though theoretically you can do it). Instead we might notice that the x = 3 in the example (or any code following **break**/**continue**) will never get executed!

Can you implement this short cut?

2.7 Paperwork

What is the grammar of SMALL now? Can you incorporate the changes you've made?

2.8 An Optional Problem: for-loop

Here is a syntax definition for a for-loop in SMALL

forLoop : **for** assignmentList [**while** expression] [**then** assignmentList] **do**
statementList **end**

e.g

```
for x = 3 ; y = 1  while x > y then x – ; y++ do
        print x, y
end
```

assignmentList is created the same way as statementList but contains only assignment type statements (including prefix INCREMENT and DECREMENT if you've implemented them).

This looks complicated but it is really just about joining bits together. The example is almost the same as:

```
x = 3
y = 1
while x > y do
        print x, y
        x - -
        y++
end
```

while and **then** are optional, what does the code become if one or both aren't present.

3 Semantic Analysis

In a large, complex compiler dealing with data types is difficult. In C++ for example you can redefine '+' and give it a meaning for every combination of types imaginable. Java offers much less freedom, though you can use '+' for concatenating strings and adding numbers or characters to strings.

I want you to work in groups of 3 or 4 people to add strings to SMALL as proper values including string variables, and expressions. If you really want to work alone you may do so but you may find you cannot do as much.

Some of the code needed is always in the small SMALL compiler, for other parts you will have to make your own changes. The 'correct' way of doing this is to create an AST without worrying whether expressions make sense (e.g "abc" - "def" might be accepted during parsing). When parsing is complete, the compiler 'walks' (traverses) the AST and adds extra information. For example '==' is a different operation on strings from on integers; semantic analysis might add information to show which equality test is needed.

Here you'll use a slightly simpler approach: type checking will be done mostly during code generation. This is less efficient but that doesn't matter here.

3.1 Specification

The aim is to add integer and string variables in SMALL with basic operations like converting string to integers and *vice versa.* Because of the way SMALL was written we won't declare variables to be of type integer or string but use an idea from BASIC: variables holding strings will have names ending in $, e.g. x$ will be a variable holding a string, x will be a different variable holding an integer.

The new, improved SMALL will have the following features:

1. **`str`** an operator to convert an integer to a string. For example you could have `var$ = "It is " + str(2017)` ('+' will works with strings). **`str`** is a unary operator and doesn't strictly need ().
2. **`int`** is the reverse `int "2017"` and returns the number `2017`. If the string isn't an integer a very large negative number is returned. That could change.
3. `>>` `z$ >> 3` will return the last three characters of `z$` as a string. So if `z$` contains `"hello world"`, `z$ >> 3` returns `"rld"`.
4. `<<` `"hello" << 4` returns the first characters `"hell"`
5. `+` can be used to concatenate strings as in java. `"hello" + " " + "world"` is `"hello world"`. As in java if either the left or right hand item is an integer, it is converted to a string.
6. `<, >=, >, <=, ==, !=` all work on strings. `string$ == "hello"` means the same as .equals() in java. This should work automatically and you need not make any changes.

7. **`len`** applied to a string returns it's length `len "xyz"` is 3.
8. You can only assign an integer value to an integer variable. Assigning a integer value to a string variable converts it to an string. i.e. `x$ = 1+ 2 * 3` makes x$ equal to "7".
9. ++ and - - cannot be used on string variables.

Error messages should be produced where appropriate. Because of the poor design of SMALL (all my fault), error messages during semantic analysis/code generation don't include line numbers.

As mentioned above, you should work in a group. If you work on the same files make sure your changes don't get out of step. If you know how, you could use version control. If not be careful and tell each other what you are doing.

3.2 How to do it

A number of features involved in string handling are already in the compiler. In a number of places the compiler already behaves differently when it sees a name ending in '$'. But up to now that never happened. However a number of changes are necessary to make the whole thing work.

The following tasks are involved:

1. Update Token.java.
    1. In the definition allow '$' to be included in identifiers. You can either include '$' in the set of legal characters or put +maybe(DOLLAR) on the end of the definition.
    2. Define patterns for TO_INT, TO_STR and LEN_STR.
2. Update Parse.java.
    1. Recognise string literals as part of expressions (see term()).
    2. Include **int, str** and **len** as unary operators (in term(), see unary '-' as an example).
    3. This has an impact on **print**. In the current version printStatement() checks for a string in the input as a special case, in the new version it is automatically handled by calling expression. The special case needs to be removed.
3. Look at CodeGen.java:
    1. In code for ASSIGN, if you try to assign a integer value to a string variable, you get an error message. It would be better to use the **str** operator to make a conversion, so x$ = 3 becomes equivalent to x$ = **str** 3.
    2. After the line containing `//!!! Insert String operations here !!!` insert code to generate the operations **int, str** and **len**. Look at NEGATE for guidance. The variable `child0IsString` tell you what type of value is on top

of the stack at this point. Trying to apply an operator to the wrong type of data should produce an error message.

Finally, a value of STR_TYPE is returned if the result stack top is, or has been converted to, a string, INT_TYPE for an integer. Remember return the correct type for all changes.

3. Find the line containing `// !!!!! STRING OPS NOT YET COMPLETE !!!!` Code for both operands has been produced. If the input was, for example, `x + 1 then` code to load `x` (child0) onto the stack was output followed by code to load `1` (`child1`). A call to emit(PLUS) will generate code to add them. But it will only work if they are both integer. Here are the possibilities

child0IsString	child1IsString	Example	Action
false	false	4 + 5	emit(PLUS);
false	true	4 + “cat”	see below
true	false	“cat” + 4	see below
true	true	“cat” + “dog”	emit(CONCAT);

The first and last cases are easy. The token name CONCAT produces the correct result for two stings as PLUS does for two integers. The third case isn’t difficult either: `emit(TO_STR)` will convert the top of the stack to a string i.e. “4” then the two strings can be concatenated. One operation which can be useful here is SWAP. emit(SWAP) produces code to swap over the top two stack values, convert the int to a string and swap them back.

You can deal with << and >> in the same way except that no conversions are needed. The string version of << it is LEFT_STR and for >> it is RIGHT_STR. emit(LEFT_STR) for the first, emit(RIGHT_STR) for the other.

4. Test it all!

4 Assessment

The assessment will be based on the course work. It should be submitted by email to me at simon@ouryard.net. I will send you an acknowledgement when I receive it.

The email should have as an attachment a zipped (.zip or whatever you prefer) copy of the the whole SMALL folder containing the the changes you have made to the files. Some files will not have changed: submit them anyway.

The assessment will be based on the following criteria:

1. Coursework section 1: 10% if you have completed at least one problem.
2. Coursework section 2: 25% if you have completed at least one problem.

3. Coursework section 3: 60% for a completed version.
4. A grammar for your enhanced version of SMALL: 5%. The grammar should follow the basic structure of that in (SMALL.pdf). You need not define the terms Name, StringLiteral or Number in the grammar but should include: a proper grammar for Expression showing the ‘levels’ (you can ‘back port’ most of this from Parse.java) and the additions you have made. It should be included in the SMALL folder and may be in any common file format (PDF, Word, ODT, text etc).

Marks for the first three sections will be assessed by building the compiler and running test data through it. I will not be assessing the quality of your coding – just seeing if it works.

It is better to have a program which is incomplete but works in some ways than one which is nearly complete but doesn’t compile.