CS2001 Week 5 Practical
Context-free grammars

130016030
University of St Andrews
October 17, 2014

Introduction

This practical requires to design a context-free grammar and a parsers that are common tools for many applications. "While they can be built by hand, they are more usually built using tools that take a description of a grammar and either interpret it or (more commonly) build code to parse an input string entered according to the grammar. Industrial-strength parser generators typically build table-driven parsers." However, in for this assignment is perfectly sensible to design recursive-desent parser.

Grammar Design

First of all, it is important to define and design a context-free grammar.

"Program" → *"StatementList"*
"StatementList" → *"Statement"* | **;** *"StatementList"*
"Statement" → *"While"* | *"For"* | *"If"* | *"Assignment"* | *"Print"*
"While" → **while** *"LogicalExpression"* **do** *"StatementList"* **end**
"For" → **for** *"Assignment"* **;** *"LogicalExpression"* **;** *"Assignment"* **do** *"StatementList"* **end**
"If" → *"LogicalExpression"* **then** *"StatementList"* **end**
| | *"LogicalExpression"* **then** *"StatementList"* **else** *"StatementList"* **end**
"Assignment" → *"Variable"* **=** *"Expression"*
"Expression" → **(***"Expression"***)**
| | *"Variable"* (**+**|**-**|*****|**/**) *"Expression"*
| | *"Variable"*
"LogicalExpression" → **(***"LogicalExpression"***)**
| | *"Variable"* (**&&**|**||**|**==**|**<**|**>**) *"Variable"*
"Variable" → *identifier*

This grammar describes "Java-ish" programming language. I believe that the grammar syntax is readable and makes sence. Assuming that I would like to provide and explain several assumption that has been made in order to complete this grammar. One of them is an identifier in variable. Identifier is a string that begins with a letter followed by 0 or more letters and digits. Another one is the syntax in For statement. Since the syntax in practical specification is not defined I assumed that there is no need for brackets and it contains assignment, then logical expression, then another assignment. I assume that the first assignment is declaraion, for example: i = 0; and second one is changing is, for instance: i = i + 2. As you can see here, it is an extensions. Another extension is While statement, which is trivial. Also, it has been decided to add recognition without ELSE in If statement. On top of that, additional logic opperators, such as: &&, || has been added in logical statement.

Recursive-desent parser

```
k = input
j = 1 + k * 3
print k, j
x = 10
if x > 10 then print 10; x = 2 else x = 3 end
if x < 5 && x > 1 then print x end
while x < 10 do prind j
Missing "end" at the end of the WHILE statement
while x < 10 do print k end
quit

Process finished with exit code 0
```

This is a typical session in the program. If the statement is illegal, it will print an error message and it would not print anything otherwise.

According to the grammar, it has been decided to create classes, namely: **Assignment, Expression, LogicalExpression, Statement and Variable.**

Assignment.java parses assignment by spliting it by equal sign, into the varible and expression. The parsing of them is done by calling methods in such instances. The equal sign always will be in this statement. Error handling of this in Statement class.

Expression.java parses expression by removing the brackets and spliting it into the variables by arithmetic symbols. Sophisticated bracket removal might be counted as another extension.

```
x = (3 + ( (2 - (10 + 4))+3) * 5); x = 20 + 3;
(3 + ( (2 - (10 + 4))+3) * 5)
3 + ( (2 - (10 + 4))+3) * 5
(2 - (10 + 4))+3
(2 - (10 + 4))
2 - (10 + 4)
10 + 4
20 + 3

Process finished with exit code 0
```

In this syntax it is not important to separate all tokens by space. This print is additional to show how recursive calls works. As you can see, if the bracket is at the beginning and at the end, they will be removed and expression will be called again. If the bracket is only at the end or only at the beggining, statements that are out of the brackets will be splited by arithmetic symbols.

LogicalExpression.java extends Expression but splits values by logical operators. Also, logical expressions must be even, for example: == instead of =, || instead of |. To recognize valid syntax there is an additional method that ensures it.

Statement.java splits statements by a semicolon. After that, it evaluates a feature of the language by checking the first token. If there are no such cases, then the statement is illigal and error message will be displayed.

Variable.java ensures that a token begins with a letter followed by 0 or more letters and digits.

Testing

Testing comes to be one of the most importnat part of the practical. Implement the robust application is twice more important.

```
if then else end
Missing "logical expression" in IF statement

Missing "first expression" in IF statement
```

```
while ((x == 4 && x != 5) || x >= 6) do x = 10 + 20 end
((x == 4 && x != 5) || x >= 6)
(x == 4 && x != 5) || x >= 6
(x == 4 && x != 5)
x == 4 && x != 5
```

```
x = (1 + 2 + (3 * 4))
x = ((1)
Wrong number of brackets
x = )(
Wrong brackets order
```

```
print x , 10
print x10
x + 10
Illegal
if x < 10 do x = 23 else y = 4 end
Missing "then" in IF statement
if x < 10 then x = 23 else y = 4 end
```

```
while (x == 4 && x != 5 || x >= 6) do x = 10 + 20 end
(x == 4 && x != 5 || x >= 6)
x == 4 && x != 5 || x >= 6
```

Those screenshots illustrates error handling of the parser.

Conclusion and futher work

This practical could be potensially improved by implementing other language feature. Also, more complex error handling can be added as well as multiple expressions in brackets. It can be seen by a screenshot below that

the parser crashes if there are many different logical expressions.

```
while ((x == 4 && x != 5) || (x >= 6)) do x = 10 + 20 end
Wrong brackets order
((x == 4 && x != 5) || (x >= 6))
(x == 4 && x != 5) || (x >= 6)
x == 4 && x != 5) || (x >= 6

Process finished with exit code 0
```