Hans van der Laan
S1541587

Compiler Construction

Homework Series 1

Question 1

Coco/R:

http://www.ssw.uni-linz.ac.at/Coco/

It depends on the language-specific version. Different version for different languages are written in different languages. For example, the C version is written in C, the java version is written in Java and the C# version is written in C#.

The tool exists since at least 1985. Around that time, it was mentioned in a book *"Ein Compiler-Generator für Mikrocomputer."* by *Rechenberg, Peter* and *Mössenböck, Hanspeter*.

I could not find a complete java grammar written in this language. There does exist an tutorial which takes java as an example to teach you about Coco/R and helps you to write a basic java grammar http://structured-parsing.wikidot.com/coco-r-parser-creating-grammar-rules-part-2.

I think the tool is on par with ANTLR. I could find a lot of information, tutorials and even books about it. It also seems well maintained, even though the last post on their webpage was done in 2014. The team consists of about 5-10 people.

JFlex:

http://jflex.de/

JFlex is written in Java.

The tool exists since 2007. The last commit was made 3 months ago, but there is a pull request waiting.

This list contains all major external JFLex grammars. https://github.com/jflex-de/jflex/wiki/External-JFlex-Grammars. It doesn't contain a JAVA grammar so we can conclude there hasn't been made one, otherwise it would've been placed on this list. This tutorial though does create a "MiniJava" grammar.

The tool seems a good tool. With just a quick google search I could find a lot of tutorials.
There are 4 collaborators. A fifth person has made a contribution to the github project, but that contribution consists of only 12 added lines and 2 removed lines so I don't consider her a full collaborator.

JavaCC

https://javacc.java.net/

JavaCC is written in Java.

I couldn't find an exact year in which JavaCC was created, but the first mention I found about it is from a JavaWorld article around 2000. The last stable release is from 2014.

There exist a JAVA Grammar written in JavaCC. http://mindprod.com/jgloss/javacc.html

It seems useable, but old and not well maintained anymore. Their webpage is just static html and most of the tutorials are more then 2-3 years old. I have the feeling that at the moment nobody is actively working on it.

SableCC

www.sablecc.org

SableCC is written in Java

The tool exists since 2008. The last commit in their repository was from 11 months ago.

There are multiple java grammars for SableCC: http://sablecc.sourceforge.net/grammars.html.

The tool seems relatively well maintained, but it seems like a relatively small project compared to the other 3 projects mentioned above. Their webpage doesn't contain a lot of information and I couldn't find as many high-quality tutorials as with the other 3 tools mentioned above. There are 5 collaborators.

CookCC

https://dzone.com/articles/cookcc-unique-lexerparser-gene ‘

CookCC is written in Java.

The tool exists since 2008. It seems pretty well maintained, with 41 commits in the last year and one bugfix in April.

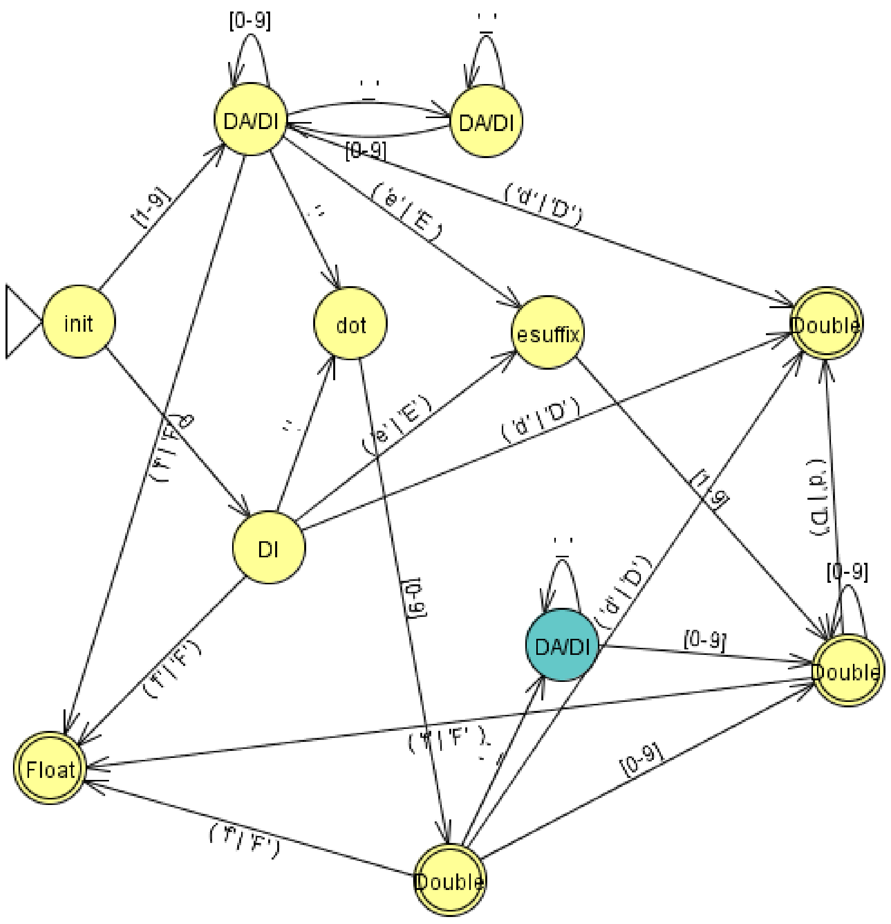
I could not find a Java Grammar written in the input format of this parser generator.

I couldn't find many good tutorials on it, and there is only one contributor. I don't think this is a very useable tool.

Question 2

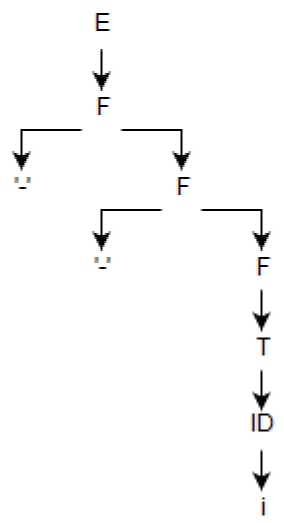
1.
They have changed it because otherwise it would allow for parsing ambiguity. This is because the e is a valid hex digit. For example: 0x1e+4. Is that a hex double or the sum of two integers, 0x1e and 4? When we change the e to p, the ambiguity is resolved. (0x1p+4)

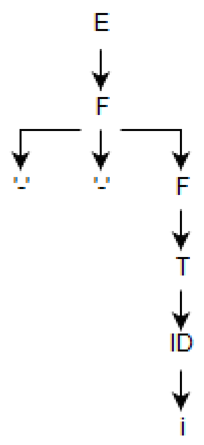
3.



Question 3

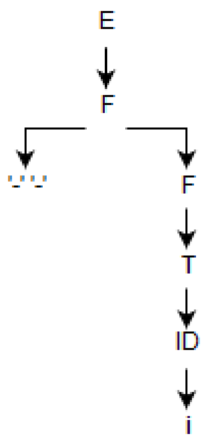
1. Example: --i





2.

Yes, it would remove the ambiguity. Now with a look ahead of one, it can choose correctly between rule 3 or 4 because they now start with different terminals.



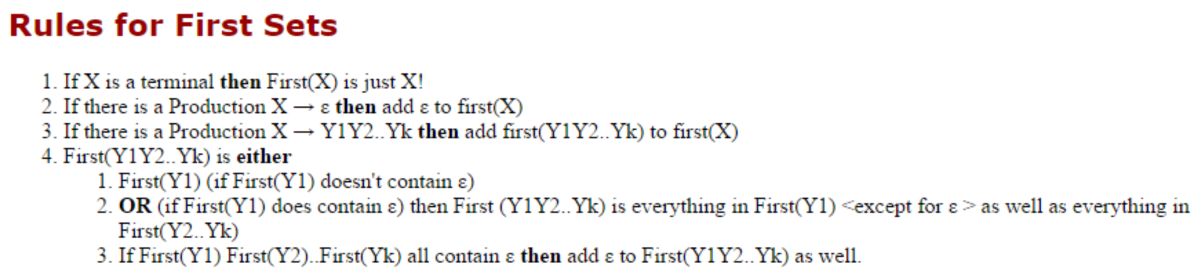
3. ANTLR produces the first parse tree (2 times '- ')
You have to switch the positions of rule 3 and 4.

4.

1	e ->	f '+' q
2	|	q
3	q ->	f '+' q
4	|	f
5	f ->	'- -' f
7	|	t
8	t ->	ID p

9		|	ID
10	p ->		‘+’ ‘+’ p
11		|	ε

5.



Rules for First Sets

1. If X is a terminal **then** First(X) is just X!
2. If there is a Production X → ε **then** add ε to first(X)
3. If there is a Production X → Y1Y2..Yk **then** add first(Y1Y2..Yk) to first(X)
4. First(Y1Y2..Yk) is **either**
    1. First(Y1) (if First(Y1) doesn't contain ε)
    2. **OR** (if First(Y1) does contain ε) then First (Y1Y2..Yk) is everything in First(Y1) <except for ε > as well as everything in First(Y2..Yk)
    3. If First(Y1) First(Y2)..First(Yk) all contain ε **then** add ε to First(Y1Y2..Yk) as well.

Let’s first list the sets we need to make :
First(e) = {}
First(q) = {}
First(f) = {}
First(t) = {}
First(p) = {}
First(‘+’) = {}
First(‘- -‘) = {}
First(ID) = {}

First, we apply rule 1
First(e) = {}
First(q) = {}
First(f) = {}
First(t) = {}
First(p) = {}
First(‘+’) = {‘+’}
First(‘- -‘) = {‘- - ‘}
First(ID) = {ID}

First, we apply rule 2 To q -> ε
First(e) = {}
First(q) = {}
First(f) = {}
First(t) = {}
First(p) = { ε }
First(‘+’) = {‘+’}
First(‘- -‘) = {‘- - ‘}
First(ID) = {ID}

We now apply rule 3, in multiple iterations

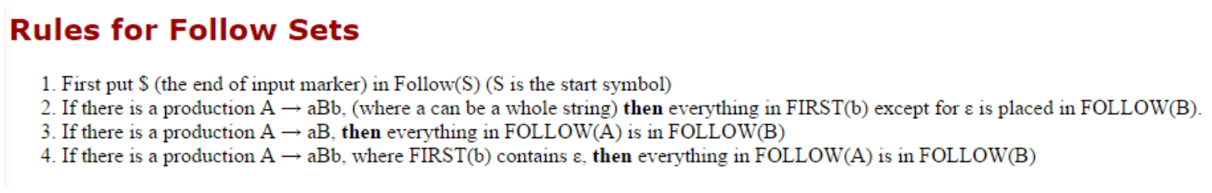
//TODO: Finish annotating the algorithm.
-Iteration 1
First(e) = {}

First(q) = {}
First(f) = {‘- -‘} // Add Tirst(t) and First(‘--‘)
First(t) = {ID}
First(p) = { ‘+’,ε } //Add First(‘+’)
First(‘+’) = {‘+’}
First(‘- -‘) = {‘- - ‘}
First(ID) = {ID}

-Iteration 2
First(e) = {‘- -‘}
First(q) = {‘--‘}
First(f) = {‘- -‘, ID}
First(t) = {ID}
First(p) = { ‘+’,ε }
First(‘+’) = {‘+’}
First(‘- -‘) = {‘- - ‘}
First(ID) = {ID}

-Iteration 3
First(e) = {‘- -‘, ID}
First(q) = {‘--‘, ID}
First(f) = {‘- -‘, ID}
First(t) = {ID}
First(p) = { ‘+’,ε }
First(‘+’) = {‘+’}
First(‘- -‘) = {‘- - ‘}
First(ID) = {ID}

-Iteration 4
First(e) = {‘- -‘, ID}
First(q) = {‘--‘, ID}
First(f) = {‘- -‘, ID}
First(t) = {ID}
First(p) = { ‘+’,ε }
First(‘+’) = {‘+’}
First(‘- -‘) = {‘- - ‘}
First(ID) = {ID}



Rules for Follow Sets

1. First put \$ (the end of input marker) in Follow(S) (S is the start symbol)
2. If there is a production A → aBb, (where a can be a whole string) **then** everything in FIRST(b) except for ε is placed in FOLLOW(B).
3. If there is a production A → aB, **then** everything in FOLLOW(A) is in FOLLOW(B)
4. If there is a production A → aBb, where FIRST(b) contains ε, **then** everything in FOLLOW(A) is in FOLLOW(B)

First, we apply step 1 and place a \$ (which indicates an end of file> in the Follow(e)
Follow(e) = {\$}
Follow(q) = {}
Follow(f) = {}
Follow(t) = {}
Follow(p) = {}

Now we apply step 2

Iteration 1
Follow(e) = {$}
Follow(q) = {$}
Follow(f) = {$}
Follow(t) = {}
Follow(p) = {}

Iteration 2
Follow(e) = {$}
Follow(q) = {$}
Follow(f) = {$}
Follow(t) = {$, '+'}
Follow(p) = {}

Iteration 3
Follow(e) = {$}
Follow(q) = {$}
Follow(f) = {$}
Follow(t) = {$, '+'}
Follow(p) = {$, '+'}

Now we calculate the First+ Set
First+ (e -> f '+' q) = First(f) = {'- -', ID}
First+ (e -> q) = First(q) = {'- -', ID}
First+ (q -> f '+' q) = First(f) = {'- -', ID}
First+ (q -> f) = {'- -', ID}
First+ (f -> '- -' f) = {'- -', ID}
First+ (f -> t) = {ID}
First+ (t -> ID p) = {ID}
First+ (t -> ID) = {ID}
First+ (p -> '+' '+' p) = {'+'}
First+(p -> ε) = Follow(p) = {$, '+'}

There are 2 rules which have the same Non-Terminal on the left side and a common Terminal in the first+ set. Thus when you have a P and the look ahead sees a '+' there still is ambiguity.

6.
There are several problems:

First, the parser can't decide with a look-ahead of one if he's in the Non-Terminal e if he should follow rule 1 or 2
Secondly, the parser can't decide with a look-ahead of one if he's in the Non-Terminal q if he should follow rule 3 or 4
Thirdly, the parser can't decide with a look-ahead of one if he's in the Non-Terminal t if he should follow rule 8 or 9

There is also a problem that the non-terminal P has 2 rules which share a token in the First+ set of those rules. At the moment you are parsing top-down with this grammar, and we are in a Non-Terminal P and we see with LL1 a '+' we don't know which rule to follow because we can get a '+' by following rule 10 or 11 (in which case the '+' is part of rule 1 or 3)

Example:

C++

We could solve the last problem in the same way we solved the problems with the single minus and the double minuses. By creating a separate token for '+' and '++'
The rest of the problems can be solved by rewriting the grammar like this:

```
1       e ->    f q
2       q ->    '+' f q
3         |     ε
4       f  ->   '--' f
5         |     ID t
6        t  ->  '++' t
7         |      ε
```

Exercise 4 :

Attribute rules:

```
Number      ->      prf seq         Number.type <- prf.type
                                    Number.value <- seq.value
                                    seq.type <- prf.type
            |       seq             Number.type <- 10
                                    Number.value <- seq.value
                                    seq.type <- 10

Seq         ->      dig             Seq.length <- 0
                                    Seq.value <- dig.value

            |       dig seq         Seq.value <- Seq.value + dig.value *(seq.type ^ seq.length)
                                    Seq.length <- seq.length + 1
                                    seq.type <- Seq.type
Prf         ->      'x'             Prf.type <- 16
            |       'b'             Prf.type <- 2
Dig         ->      DIGIT           Dig.value <- parseHex(DIGIT)  //Deze functie bestaat
```
waarschijnlijk niet, maar het duidelijk wat hij doet en dan hoeven we niet alle mogelijkheden op te schrijven.