Compilation Techniques

Laboratory 2

Regular definitions

The regular expressions can be named and they become **regular definitions**. For the ease of writing some definitions can help in other definitions. For example, the definition of a numeric character (digit) “[0-9]” may be necessary in some places and so it can be included in other definitions. If that definition is used only inside other definitions, without forming by itself a lexical atom, we say about it that it is a **fragment**. Some software tools which help to implement lexical analyzers use the convention that the tokens and fragments start with an uppercase letter.

```
fragment DIGIT: [0-9] ;
INT: DIGIT+ ;
REAL: INT [.] INT ;
```

In the above example DIGIT is a lexical fragment, because it is used only as a part in the INT atom. INT is both a lexical atom and a part of the REAL atom definition. For regular definitions the recurrence is not allowed, so a definition cannot call (include) itself, directly or indirectly. Because of this we cannot write using regular definitions C comments “/*…*/” which are allowed to contain inside them other C comments.

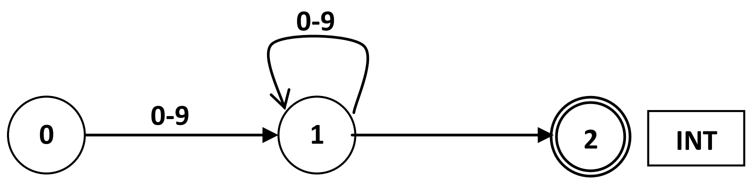
There are some more differences between regular definitions and regular expressions:

Regular definitions (RD)	Regular expressions (RE)
The spaces are not significant and the RD can be formatted so it can be more readable.	Any space is significant and it counts as input.
Because the RD can contain the name of other RD which are included, the words which are in a RD have the role of identifiers, not characters. To differentiate between them and ordinary characters, by convention identifiers can be bold or underline or, in the case of text formats, any ordinary character is put inside single or double quotes.	Any letter or sequence of letters is treated as ordinary characters, without any other meaning.

Transition Diagrams (TD)

Transition diagrams are based on the formalism of the finite automata (AF). They can be used as a way to represent RD, to help their implementation. TD are represented graphically as a directed graph which has as nodes (vertices) the possible states in which that TD can be and as edges (arcs) the transitions triggered by the input characters from the current state to a next state.

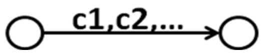
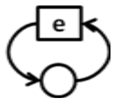
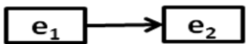
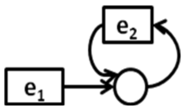
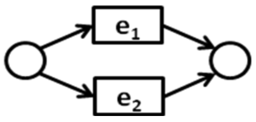
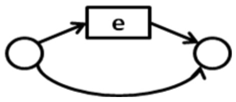
Always the start is the initial, unique state (0). When an entire RD was traversed we arrive in an accepting or final state (represented by 2 concentric circles), which means that a new lexical atom was formed using the input characters. This cycle is repeated for all input characters.



In the above example it is represented a TD of RD „INT: [0-9]+". It can be seen that starting with the initial state 0 it must be consumed at least one digit in order to reach state 1. In state 1, if there are other digits in the input stream, these are consumed and the state remains 1. The transition which has no label has the meaning of "else" or "in other case" and it is analogical to the "else" part of the if instruction. This transition is always tested last and it must not consume the input character. For example, if we have the string "25*4", the character "2" will be consumed at the transition from the state 0 to state 1 and the character "5" at the transition 1->1. Then it follows "*" which is the next lexical atom, so the recognition of all characters of the number "25" is ended and using the transition 1->2 we reach the final state 2, without consuming "*". If the transition 1->2 had consumed "*", it would have remained consumed (because in that stage only the atom "25" is returned) and on the next pass through TD the atom "4" will be recognized, so "*" would have been lost.

__Rules to construct a TD__

The representation of the principal parts of the regular expressions in TD (with rectangles are regular expressions which will be substituted as such):

Regular expression	TD
characters and characters classes – all characters or classes of characters which go from the same state and arrive in a common next state can be represented on the same transition. If their negation is necessary, they can be prefixed with ^	
e*	
e_1e_2	
e+ - it is used the equivalency e+=ee*	
e_1|e_2	
e?	

Any transition can consume at most one character (but this one can represent a class of characters). For example, to represent the beginning of single line C comments "//", two transitions must be represented, because there are two characters.

The RD which does not form meaningful tokens for the next phases of the compiler (spaces, comments) will have the final state in the initial state. In this way they can be consumed without generating tokens.

From the same state cannot originate transitions which consume the same characters and also more than one "else" transition. In this situation, if we had been in that state and a character for which many transitions are possible needs to be consumed, we would have not known what transition to follow. These cases appear when more tokens have the same prefix, for example INT and REAL which both begin with "[0-9]+". In these situations we factorize (extract the common prefix) their RD.

In many cases a definition for the end of input (\0 or EOF) is not represented but an END token is added implicitly.

If two states are linked only through an “else” state, then they can be unified, preserving in the same time all the transitions which come and go from each state.

If TD becomes too complicated because many RD must be added, these can be represented separately, by considering the initial state (0) as the common point. In this case also the above properties must be preserved. The lexical fragments and the identifiers which appear inside a RD must be expanded to their own constituents (they will not be put as names/references on TD).

At each final state (represented with double circle) it will be noted the corresponding lexical atom. The final states do not have transitions which originate from them.

Special states for lexical errors can be added, for example for when we are in a string constant (“…”) and the end of file appears before the final quote.

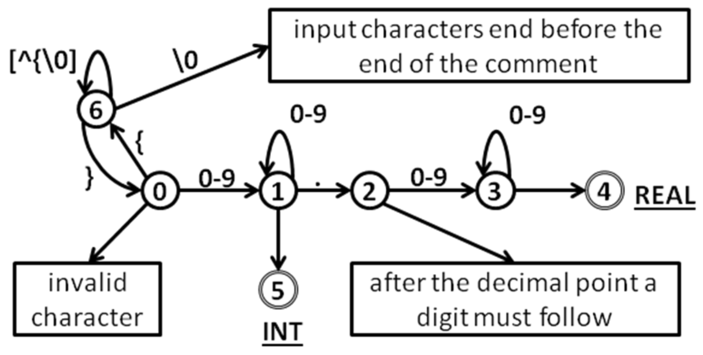
Example: represent the TD for the following RD:

fragment DIGIT: [0-9] ;

INT: DIGIT+ ;

REAL: INT [.] INT ;

COMMENT: { [^}]* } ; //does not generate a lexical atom


[^{\0]
\0
input characters end before the end of the comment
6
{
}
0
0-9
1
0-9
.
2
0-9
3
0-9
4
REAL
invalid character
5
INT
after the decimal point a digit must follow

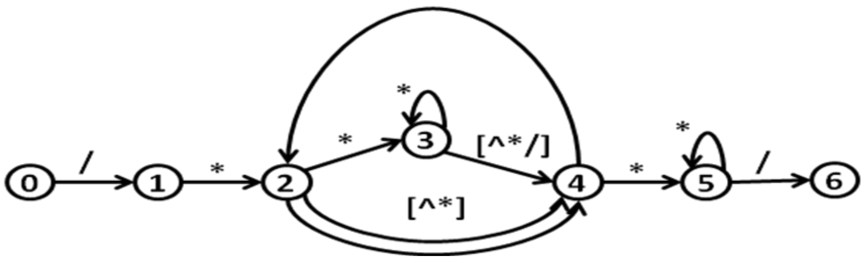
In the above example it can be seen that:

- All identifiers which appear inside other definitions (DIGIT, INT) were expanded to their own regular expressions.
- Because INT and REAL both begin with „DIGIT+”, this sequence was factorized and it became a common beginning for both RD. If after „DIGIT+” follows a dot (**.**) real numbers are considered, else integers.
- The sequence of states for comments (0->6->0) ends in the initial state, because the comments must not generate tokens.
- In certain states of TD only some characters are allowed. For example, after a decimal dot (**.**) only a digit “[0-9]” can follow. If in this state we have other input character, an error can be signaled. The errors were represented as rectangles with a descriptive text.

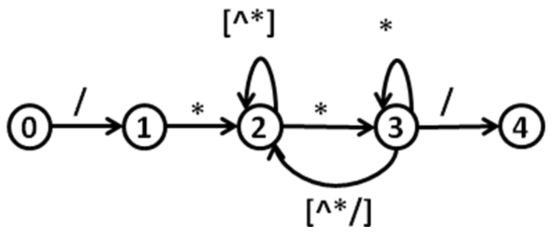
From the above considerations, TD are nondeterministic finite automata (NFA), because the “else” transition which does not consume a character corresponds to a ϵ-transition. Even in this case the rules of construction and structure for TD made possible for them to be implemented in a deterministic manner, because in any state with any char will be at most only one suitable transition to follow. In most of the times the nondeterminism can be easily solved by factorizing the common prefixes of the regular definitions. There are still cases when in RDs can be a nondeterminism which is harder to solve, especially when cycles are involved. Such an example is the definition of the C comments:

COMMENT: '/*' ([^*] | '*'+ [^*/])* '*'+ '/' ;

The rules to construct a TD from this RD generate this:



The state 6 in fact will be 0, because no token will be generated. It can be seen that the basic rules are observed (there are no two transitions with the same character and also no two “else” transitions from the same state) but still there is a nondeterminism generated by the fact that from the state 2 with “*” can result both the state 3 and the state 5, through the “else” transition 2->4. This nondeterminism cannot be directly factorized, because of the cycle 2->4->2. In this case more advanced methods must be employed to transform the NFA to DFA (Deterministic Finite Automata) and maybe to also optimize the result. An alternative would be to try intuitively to construct a DFA, by taking into account what is the purpose of that RD. The result will be something like this:



Application 2.1: *Represent the TD for the tokens CT_INT and CT_REAL from AtomC.*

Homework: Complete the TD from the application 2.1 with all the tokens from the AtomC lexical rules. The keywords will not be represented on TD, because they will be identified later as special cases of IDs.