
CS 421 – Spring 2007

Lecture Notes Set 17:

Type Inference

Elsa L. Gunter¹

Transcribed by: Pooja Mathur

Corresponding to Slides: 16 – ocamllex (slides 1 – 15)

Made available: February 28, 2007

Revision 1.0

1 Change Log

1.0 Initial Release.

2 Characters to Instructions

2.1 Leading in from last time...(slide 2)

Last time we talked about how we can figure out if a particular string is acceptable or not with DFA's and NFA's. The next step is, after figuring out if something is acceptable, to turn those strings into computer instructions.

2.2 Two Phases(slides 2 - 3)

So a computer gets in a sequence of characters and translates them sequences of tokens, which are the recognition of the words. Now we need to interpret the sequence of tokens. You can think of this being done in phases. First you get your words, then you get your sentences, then actions can be made, based off of those sentences. Initially, the sentences are going to be translated into abstract syntax trees. These trees are basically going to be breaking up the sentences to figure out what it is they are trying to say. It is similar to breaking up English sentences into nouns and verbs, to figure out what the sentence is saying. So from these abstract syntax trees, we are going to figure out what we should be doing next.

More specifically, to convert a string into an abstract syntax tree, first comes lexing. That is where you convert strings into lists of tokens or 'words of the language'. Then there is parsing, where the list of tokens is fed into the abstract syntax tree.

3 Lexing

3.1 A quick example (slide 4)

When lexing, one has to figure out what each set of characters is trying to get at. That is, if you saw the characters 'asd', you that would consider that a string. If you saw the characters '123', then that would be an integer. The sequence '3.14' would be a float. Also, when trying to interpret when to divide a sequence of characters into parts, there has to be some indicator used. Typically, white spaces signify that a word has ended and another one can begin. These white spaces include, spaces, tabs, new lines, and character returns. Then there are delimiters, that can group sets of characters together, but they aren't as simple as white spaces, delimiters have meaning too.

¹© 2007, Share and Enjoy

3.2 How lexing comes in (slide 5)

For lexing, we want to be able to specify what our words are. We want to have a specification for what a word can be, and that is where regular expressions come in. With those, we can tell if what we have seen belongs in our program. We will use these regular expressions to describe what our identifiers are, for example. Also, we want to have an efficient way of doing these translations from characters to words. The program that does this, that takes these strings and generate our DFA's to do the translations is called lex.

But there is a problem, we want to be able to pull out a bunch of words from a string, not just one word per string...what do we do? Well what we want is for there to be a sequence of DFA's so we can let the string run through everything and then be figured out. We also want to be able to know when we have reached a final state for a word, even when there are more characters in our string.

3.3 Lexing on strings of words (slide 6)

So, when given a whole string of words to split up we need to take a modified DFA. When we get to a white space, we have reached the end of a word, and there we check if we are in an accept state, if so then return that token and parse the rest of the string. Otherwise, we did not find a word and we should stop. So we do this recursively, until we have either finished the whole string, or we have ended some word in a non-accept state (meaning the 'word' does not fit in our language).

3.4 DFA example (slide 7)

So with the DFA in the slide, we start at the start state, and if we see a sequence of letters, and then a white space, we can say that we have seen a string. If we have seen a bunch of numbers in a row, then we have seen an integer. But if we have seen a bunch of numbers, a decimal and then more numbers, that would be a float. Each of the three states is a different accept state that means a different word was seen, string, integer, and float.

4 OCamllex

4.1 The program (slide 8)

Now, we could do write the regular expressions and them to DFAs by hand...but that would take a lot of work. Why not write a program that takes in the regular expressions as input and generates the automata for us. OCamllex is what does this for OCaml. OCamllex produces a program of DFA's, more or less, that interprets strings of characters into words based on the input regular expressions.

4.2 How the program works (slides 9 - 10)

First and foremost, you need the set of regular expressions that are acceptable to the language. You can't do anything without those basic rules. One other thing we need for the program to work is to have some buffer that takes in the string that we want to interpret. This is called the lexing buffer. Then you also need to have the corresponding set of actions taken when a sequence of characters are matched. That is, when you end at an accept state, what do you do with that 'word'?

Then, the lexer will take the regular expressions it was given. It will create the state machine. The state machine will take in the lexing buffer and 'play the state machine game' that we talked about last time, and if we reach an accept state, the program will perform some action that was specified for it.

4.3 How to use the program (slide 11)

Basically, the way to use this is first you put a table of regular expressions the the actions associated with the accept states into a file with a .mll extension (the extra 'l' is for lexer and will be taken off by OCaml). The you run:

```
ocamllex <filename>.mll
```

This produces the OCaml code that will, more or less, run the state machines and perform the actions you specified. This goes into a .ml file with the same file name. If <filename>.ml already existed before, it will be over written.

4.4 Sample input (slides 12 - 14)

So, to start off, you use the 'rule' keyword, to signify that you are writing a rule. Then after that, you write the name of your program. Here, we used main. You don't have to use main, the only reason we did is because it is considered to be traditional in a sense. But there is no need to use it.

After that we have the keyword parse and then the set of regular expressions, separated by or's. It is basically going through the DFA example from slide 7. That is, if we see any of the characters from 0 through 9 in a sequence and then a white space, that is an integer and the action to be taken is to print the string 'Int'. Then if we see a sequence of characters 0 through 9 a decimal and then another sequence of characters 0 through 9, that is another accept state and the action is to print the string float. For strings of letters, we follow the same idea.

More technically, for a rule, you have a square bracket, the sequence that is allowable. A single character doesn't need brackets around it. For the allowable sequence, the start and end of that sequence needs to be in single quotes. Then the end square bracket. If there is a plus sign after the character, or sequence of characters, it means that if you were to draw the DFA there would be an arrow pointing to itself. In other words the elements in the sequence can be repeated. After the acceptable sequences, you have a curly brackets and then inside the curly brackets are OCaml code for what you want to happen if that sequence was seen. For the last or, we have the wild card underscore. That is basically saying that if we get something in the string that doesn't fit any of the other types, we'll go here, and the action associate with it is to call main again on lexbuf. But, remember, every time we see a character, we throw it away, so lexbuf has been getting smaller as the program has been running.

```
rule main = parse
  ['0'-'9']+ {print_string"Int"}
  | ['0'-'9']+ '.' ['0'-'9']+ {print_string"Float"}
  | ['a'-'z']+ {print_string"String"}
  | _ {mainlexbuf}
```

This part of the code does a little set up. It basically sets up the lexing buffer and then most importantly, it calls main.

```
let newlexbuf = (Lexing.from_channel stdin) in
print_string "Ready to lex.";
main newlexbuf
```

So in general terms, you have at the beginning of your file, basic information for your lexer. That is, if you need to define any data types or open any files, you would do this at the top. Then you have the rule keyword, the program name, and then the arguments of the program. Note that lexbuf is an implied argument, so you don't have to type it. Then the keyword parse is needed. After that, you have or statements with that go regular expression and then action in curly brackets. Also, you can have multiple lexers in the same file that are mutually recursive. So that they call each other. After all that you will have your trailer, which is basically, creates the lexing buffer and calls your program.