MAU22C00 Lecture 31

John Stalker

Trinity College Dublin

Context free languages

What makes a language context free is that

- it has a phrase structure grammar
- the possible expansions of a non-terminal symbol don't depend on its context, i.e. on the surrounding symbols

Natural languages are almost never context free.

Programming languages tend to be context free-ish. There's a parsing phase, based on a context-free grammar, but not every program which is parsed successfully is valid, because of context sensitive rules like "all variables must be declared before use".

Badly designed programming languages may not even have a context free grammar at the parsing stage.

Data languages are often truly context free.

Examples

- All regular languages are context free, since right and left regular grammars are context free grammars.
- The language of balanced parentheses is context free, since it has the grammar

```
%%
okay : | ( okay ) okay ;
```

  It is not regular though.
- The language of palindromes is context free. I gave a grammar for it in Lecture 2. It's also not regular.
- The languages of strings matching the regular expression `x*y*z*` with the same number of x's as y's is context free. There's a grammar in the notes. Without the additional restriction it would be regular, but with it it's not. The same applies if we require instead that there are the number of y's as z's. What if we require both?

Pushdown automata

A pushdown automaton is like a finite state automaton with a stack.

The stack is initially empty.

Just as we can check whether the input is exhausted, and read a token if there is one, we can check whether the stack is empty, and pop an item if there is one.

We can also push items onto the stack though.

The items on the stack could be from the same set as the tokens of the input, but don't have to be.

As with FSAs, a pushdown automaton could be deterministic, but we're not assuming it is unless we say so explicitly. The usual notion of success applies.

There are two notions of "state" for a pushdown automaton: a narrow notion, without the stack, and a wide notion, with the stack. The narrow state is restricted to a finite set, but the wide state is not.

Reading a token or popping an item can induce a (narrow) state transition or stack operations.

Example (balanced parentheses)

I gave a pushdown automaton recognising the language of balanced parentheses in Lecture 3. Here's a slightly simpler one.

- If the input is exhausted we terminate, successfully if the stack is empty and unsuccessfully if it is non-empty.
- If we read a (we push a).
- If we read a) we check whether the stack is empty. We terminate unsuccessfully if it is and pop the top item, which is always a), if the stack is non-empty.

This pushdown automaton happens to be deterministic.

There are two ways to think about the stack contents:

- The stack is a simple counter. Pushing increments. Popping decrements. The stack size tells us how many of the ('s we've seen have yet to be matched. Which item we use for this purpose is irrelevant.
- The stack gives a member of the set of valid (right) continuations of the input. In particular it gives us the shortest such continuation.

More examples

- The pushdown automaton from Lecture 3 kept a representative of the current bidirectional equivalence class on the stack.
- A pushdown automaton recognising the language of zeroeth order logic is given in the notes. It's deterministic and employs the same trick of keeping a valid continuation on the stack. This trick does not work for all context free languages though.
- It is possible to construct a pushdown automaton for all context free languages though, by keeping a list of symbols on the stack.

From context free grammars to pushdown automata

The pushdown automaton associated to a phrase structure grammar begins by pushing the start symbol.

At each step it checks whether the stack is empty and pops of the top item if not. What happens next depends on whether it was empty and whether that item was terminal.

- If the stack was empty it terminates, successfully if the input is exhausted and unsuccessfully if there is more input waiting.
- If the stack was non-empty and the top item was terminal it tries to read an input token and terminates successfully if the token belongs to the symbol. Otherwise it terminates unsuccessfully.
- If the stack was non-empty and the top item was non-terminal it chooses an alternate and pushes those symbols in reverse order, so the first symbol in the expansion is the last pushed, i.e. the new top item on the stack.

This last step makes the automaton non-deterministic if there is a rule with more than one alternate.

Examples

To see this in action for the language of zeroeth order logic see the notes.

What happens if we apply this construction to a right regular grammar?

The following is a right regular grammar for the language of integers, based on the strongly deterministic finite state automaton from last lecture:

```
%%
integer : A0 ;
A0 : - A2 | 0 A1 | 1 A3 | 2 A3 | 3 A3 | 4 A3 | 5 A3 | 6 A3 | 7 A3 | 8 A3 | 9 A3 ;
A1 : | - A4 | 0 A4 | 1 A4 | 2 A4 | 3 A4 | 4 A4 | 5 A4 | 6 A4 | 7 A4 | 8 A4 | 9 A4 ;
A2 : - A4 | 0 A4 | 1 A3 | 2 A3 | 3 A3 | 4 A3 | 5 A3 | 6 A3 | 7 A3 | 8 A3 | 9 A3 ;
A3 : | - A4 | 0 A3 | 1 A3 | 2 A3 | 3 A3 | 4 A3 | 5 A3 | 6 A3 | 7 A3 | 8 A3 | 9 A3 ;
A4 : - A4 | 0 A4 | 1 A4 | 2 A4 | 3 A4 | 4 A4 | 5 A4 | 6 A4 | 7 A4 | 8 A4 | 9 A4 ;
```

We start by pushing `integer`, then immediately popping it and pushing `A0`.

What happens next depends on the input. Suppose the input is `2023`.

Example, continued

```
A0 : - A2 | 0 A1 | 1 A3 | 2 A3 | 3 A3 | 4 A3 | 5 A3 | 6 A3 | 7 A3 | 8 A3 | 9 A3 ;
A1 : | - A4 | 0 A4 | 1 A4 | 2 A4 | 3 A4 | 4 A4 | 5 A4 | 6 A4 | 7 A4 | 8 A4 | 9 A4 ;
A2 : - A4 | 0 A4 | 1 A3 | 2 A3 | 3 A3 | 4 A3 | 5 A3 | 6 A3 | 7 A3 | 8 A3 | 9 A3 ;
A3 : | - A4 | 0 A3 | 1 A3 | 2 A3 | 3 A3 | 4 A3 | 5 A3 | 6 A3 | 7 A3 | 8 A3 | 9 A3 ;
A4 : - A4 | 0 A4 | 1 A4 | 2 A4 | 3 A4 | 4 A4 | 5 A4 | 6 A4 | 7 A4 | 8 A4 | 9 A4 ;
```

Our input is still `2023` and our stack is `A0`.

We pop the `A0` and push one of its alternates, e.g. `- A2`.

`-` is terminal, so we read the first token, a `2`, see that they differ, and fail.

This failure is just a local failure though. Some other computational path might succeed.

We could have chosen `2 A3` instead, for example. Then we would have pushed `A3` and then `2` onto the stack, so it would be `2 A3`.

Example, continued

```
A0 : - A2 | 0 A1 | 1 A3 | 2 A3 | 3 A3 | 4 A3 | 5 A3 | 6 A3 | 7 A3 | 8 A3 | 9 A3 ;
A1 : | - A4 | 0 A4 | 1 A4 | 2 A4 | 3 A4 | 4 A4 | 5 A4 | 6 A4 | 7 A4 | 8 A4 | 9 A4 ;
A2 : - A4 | 0 A4 | 1 A3 | 2 A3 | 3 A3 | 4 A3 | 5 A3 | 6 A3 | 7 A3 | 8 A3 | 9 A3 ;
A3 : | - A4 | 0 A3 | 1 A3 | 2 A3 | 3 A3 | 4 A3 | 5 A3 | 6 A3 | 7 A3 | 8 A3 | 9 A3 ;
A4 : - A4 | 0 A4 | 1 A4 | 2 A4 | 3 A4 | 4 A4 | 5 A4 | 6 A4 | 7 A4 | 8 A4 | 9 A4 ;
```

The remaining input is currently still `2023` and the stack is `2 A3`.

`2` is terminal so we read an input token, which is also a `2`, so we're still okay. The remaining input is `023` and the stack is `A3`.

We choose an alternate for `A3`. We could choose the empty alternate, for example. If we do then at the next step the stack is empty, so we check for more input, find that there is some, and fail.

This failure is again just a local failure though. Some other computational path might succeed.

We could have chosen `0 A3`, for example. `0` is terminal so we read an input token, which is also a `0`, so we're still okay. The remaining input is `23` and the stack is `A3`.

Example, continued

```
A0 : - A2 | 0 A1 | 1 A3 | 2 A3 | 3 A3 | 4 A3 | 5 A3 | 6 A3 | 7 A3 | 8 A3 | 9 A3 ;
A1 : | - A4 | 0 A4 | 1 A4 | 2 A4 | 3 A4 | 4 A4 | 5 A4 | 6 A4 | 7 A4 | 8 A4 | 9 A4 ;
A2 : - A4 | 0 A4 | 1 A3 | 2 A3 | 3 A3 | 4 A3 | 5 A3 | 6 A3 | 7 A3 | 8 A3 | 9 A3 ;
A3 : | - A4 | 0 A3 | 1 A3 | 2 A3 | 3 A3 | 4 A3 | 5 A3 | 6 A3 | 7 A3 | 8 A3 | 9 A3 ;
A4 : - A4 | 0 A4 | 1 A4 | 2 A4 | 3 A4 | 4 A4 | 5 A4 | 6 A4 | 7 A4 | 8 A4 | 9 A4 ;
```

The remaining input is 23 and the stack is A3.

We choose the expansion 2 A3 again and successfully match the terminal 2 against the next input token so now the remaining input is 3 and the stack is A3.

Next we choose the expansion 3 A3 and successfully match the terminal 3 against the next input token so now the remaining input is empty and the stack is A3.

Now we choose the empty expansion, so we have an empty stack and empty input and this computational path succeeds.

Since there is *some* computational path which succeeds the input is recognised, and 2023 is an integer.

Comments

Our grammar was unambiguous but the pushdown autmaton is still non-deterministic. Ambiguity and non-determinism are different things.

This pushdown automaton only ever uses the top two positions on the stack.

That's true for every automaton constructed from a right regular grammar.

Had we used the left regular grammar for this language there would have been no upper bound on how large the stack could grow.

If we have a pushdown automaton which can work with a finite stack we can always find a finite state automaton which recognises the same language, so it is a regular language.