CS 241 — Lecture 14

Bartosz Antczak *Instructor: Kevin Lanctot* *February 16, 2017*

Recall — Ambiguous Grammar

Since grammar can be ambiguous (i.e., "$9 + 3/3 = 4 \,\text{or}\, 10$?), we can have multiple parse trees for the same expression. The resulting string from a parse tree depends on how we *traverse* it. To make it unambiguous, we need to have a more formal set of production rules:

- $\alpha A\beta$ *directly derives* $\alpha\gamma\beta$ if there is a production rule $A \to \gamma$, where:
  - $A \in N$ (non-terminals), and
  - $\alpha, \beta, \gamma \in (N \cup T)$ (non-terminals, terminals, empty string)

  Informally, "directly derives" means it takes one derivation step or one application of a production rule.

- $\alpha A\beta$ *derives* $\alpha\gamma\beta$ if there is a finite sequence of productions $\alpha A\beta \to \alpha\Theta_1\beta \to \alpha\Theta_2\beta \to \cdots \to \alpha\gamma\beta$, where again:
  - $A \in N$ (non-terminals), and
  - $\alpha, \beta, \gamma \in (N \cup T)$ (non-terminals, terminals, empty string)

  It is written as $\alpha A\beta \implies {}^*\alpha\gamma\beta$

To reduce ambiguity, we will set up some standard for reading strings:

- **Associativity:** how we evaluate symbols (e.g., $6 - 3 + 4$: do we read it as $(6 - 3) + 4$ or $6 - (3 + 4)$?). We will set a standard for *left associativity*
- **Precedence:** grouping non-equivalent terminals. For instance, in arithmetic, multiplication takes precedence over addition.

14.1 Top-Down Parsing

Parsing is the approach to determining if a certain string is valid in a given grammar. In other words, given a grammar G and a word w, *find a derivation for w.*
Our goal in this section is to look at the characters in w and decide which rules derived w from the start symbol.

14.1.1 Approach 1 — Backtracking

We can use a *backtracking algorithm* for parsing, which considers every possible derivation and stops once it finds a valid one. This approach is very exhaustive, so let's try another approach.

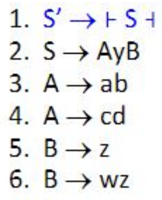
14.1.2 Approach 2 — Stack-based Parsing

Use a stack to remember information about our derivations and/or processed input. To use stack-based parsing, we must also *augment* our grammar by adding a two new characters: one to mark the beginning of a word (called BOF) and one to mark the end of a word (called EOF). We must also define a new start symbol S' that only appears once as $S \to startSend$ .The method works like this:

- Start by pushing S' on to the stack
- For every item on top of the stack:
  - If it is a **non-terminal**, expand it using a production rule where the resulting string matches the input
  - If it is a **terminal**, match it with the input. Pop the terminal off the stack and read the next character from the input

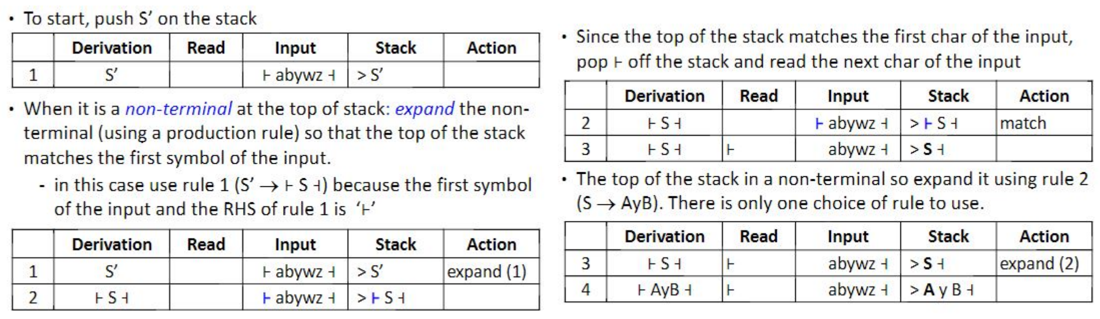
If you try to pop when the stack is empty, or when the stack is not empty when you are finished processing, that results in an ERROR.

Example 14.1.1. *The first few steps in deriving "abywz" from a given grammar (the remaining steps follow the same procedure as the first few steps and were omitted to save space)*



1. S' → ⊢ S ⊣
2. S → AyB
3. A → ab
4. A → cd
5. B → z
6. B → wz

Figure 14.1: The production rules for this particular grammar example. Courtesy of Prof. Lanctot's slides.



- To start, push S' on the stack

	Derivation	Read	Input	Stack	Action
1	S'		⊢ abywz ⊣	> S'	

- When it is a *non-terminal* at the top of stack: *expand* the non-terminal (using a production rule) so that the top of the stack matches the first symbol of the input.
  - in this case use rule 1 (S' → ⊢ S ⊣) because the first symbol of the input and the RHS of rule 1 is '⊢'

	Derivation	Read	Input	Stack	Action
1	S'		⊢ abywz ⊣	> S'	expand (1)
2	⊢ S ⊣		⊢ abywz ⊣	> ⊢ S ⊣	

- Since the top of the stack matches the first char of the input, pop ⊢ off the stack and read the next char of the input

	Derivation	Read	Input	Stack	Action
2	⊢ S ⊣		⊢ abywz ⊣	> ⊢ S ⊣	match
3	⊢ S ⊣	⊢	abywz ⊣	> **S** ⊣	

- The top of the stack in a non-terminal so expand it using rule 2 (S → AyB). There is only one choice of rule to use.

	Derivation	Read	Input	Stack	Action
3	⊢ S ⊣	⊢	abywz ⊣	> **S** ⊣	expand (2)
4	⊢ AyB ⊣	⊢	abywz ⊣	> **A** y B ⊣	

Figure 14.2: The stack approach. Courtesy of Prof. Lanctot's slides.

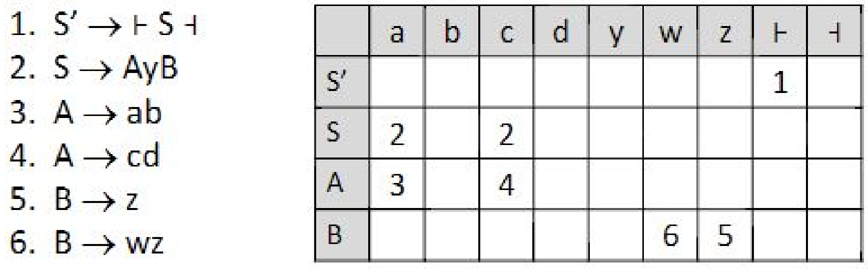
Now that we know how this works, one problem still stands: how are we able to correctly predict which rule applies?

14.1.3 LL(1) Parsing

LL(1) parsing is named after:

- Processing the input from **L**eft to right, finding a **L**eftmost derivation, and the algorithm allows us to look ahead **1** token.

For all LL(1) grammars, given an A on the top of the stack and an a as the next input character, at most one rule can apply. These restrictions cause our reading to be unambiguous. We do this by constructing a *predictor table.*



1. S′ → ⊢ S ⊣
2. S → AyB
3. A → ab
4. A → cd
5. B → z
6. B → wz

	a	b	c	d	y	w	z	⊢	⊣
S′								1	
S	2		2						
A	3		4						
B						6	5		

Figure 14.3: A sample predictor table. Courtesy of Prof. Lanctot's slides.

Constructing a Predictor Table

To build a method `predict(A, a)`, we require two helper methods. The method `predict(A, a)` outputs maps the input values to their respective value in the predictor table. The two helper methods are:

- `follow(A)`: this method answers the question, *starting from the start symbol, does the terminal b ever occur immediately following A?* Here, $b \in T, A \in N$
- `empty(`α`)`: answers the question, *can α disappear?* This method returns true if $\alpha \to^* \varepsilon$. It returns false if α has a terminal in it

With these helper methods, we can calculate `predict(A, a)`. Here, we're asking *if A is on the top of the stack and a is the next symbol in the input, which rule should be used to expand A?*.