Monash University
Faculty of Information Technology

Lecture 12
Parsing

Slides by Graham Farr (2017), with some by David Albrecht (2011).

FIT2014 Theory of Computation

Overview

- Definitions
- Examples
- Shift-reduce parser
- Lex & Yacc

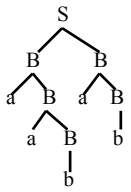
Parsing

- Suppose you have a Context Free Grammar, and a string of letters.
- **Parsing:** determining whether the string
  - is a word in the language, and if it is,
  - finding a **parse tree**, or a **derivation**, for it.
- **Parser:** a program that does this.
- Two main types:
  - **Top-down parsers**
  - **Bottom-up parsers**
    - **reduce** the string to the Start symbol
    - repeatedly apply production rules in reverse

a*ba*b

S → BB
B → aB
B → b

Input: aabab



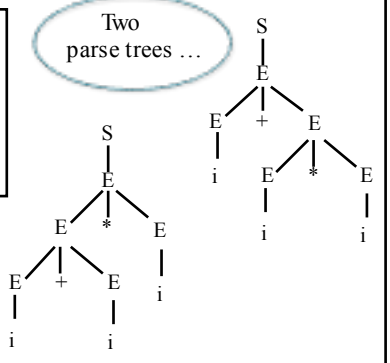
Plus-Times-A

S → E	**(1)**
E → E + E	**(2)**
E → E * E	**(3)**
E → i	**(4)**

This grammar is **ambiguous**.

Input: i + i*i

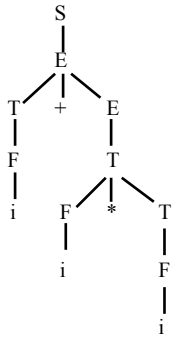
Two parse trees …



Plus-Times-B

S → E
E → T + E
E → T
T → F * T
T → F
F → i

Input: i + i*i



LR Parser

- Bottom-up Parser
- Scan input **L**eft to Right
- Construct a **R**ightmost derivation in reverse
- Implemented using a Deterministic Pushdown Automaton (DPDA).
- Not all CFGs have one: DCFL ≠ CFL
- We'll look at one type of LR parser: **shift-reduce parser**

Shift-reduce Parser

- a particular type of LR Parser
- Has:
  - a **stack:** terminals and non-terminals processed so far, and
  - a **buffer:** the rest of the input string (yet to be processed)
- Initially:
  - Stack is empty
  - Buffer contains the entire input string
- Repeatedly …
  - **Shift** input letters onto the stack, *OR*
  - When a string of top-most stack symbols equal the right-hand side of a production rule:
    - **Reduce** that string, i.e., use production rule in reverse
- … until Stack only has Start symbol, and Buffer is empty.

a*ba*b

S → BB	(1)
B → aB	(2)
B → b	(3)

Input: abb

	abb	shift
a	bb	shift
ab	b	reduce, (3)
aB	b	reduce, (2)
B	b	shift
Bb		reduce, (3)
BB		reduce, (1)
S		ACCEPT

Plus-Times-A

S → E	(1)
E → E + E	(2)
E → E * E	(3)
E → i	(4)

Input: i + i*i

	i+i*i	shift
i	+i*i	reduce, (4)
E	+i*i	shift
E+	i*i	shift
E+i	*i	reduce, (4)
E+E	*i	

*To **shift**, or to **reduce**?*

Plus-Times-A

S → E	(1)
E → E + E	(2)
E → E * E	(3)
E → i	(4)

Input: i + i*i

	i+i*i	shift
i	+i*i	reduce, (4)
E	+i*i	shift
E+	i*i	shift
E+i	*i	reduce, (4)
E+E	*i	shift
E+E*	i	shift
E+E*i		reduce, (4)
E+E*E		reduce, (3)
E+E		reduce, (2)
E		reduce, (1)
S		ACCEPT

Plus-Times-A

S → E	(1)
E → E + E	(2)
E → E * E	(3)
E → i	(4)

Input: i + i*i

	i+i*i	shift
i	+i*i	reduce, (4)
E	+i*i	shift
E+	i*i	shift
E+i	*i	reduce, (4)
E+E	*i	

*To **shift**, or to **reduce**?*

Plus-Times-A

S → E (1)
E → E + E (2)
E → E * E (3)
E → i (4)

Input: i + i*i

	i+i*i	shift
i	+i*i	reduce, (4)
E	+i*i	shift
E+	i*i	shift
E+i	*i	reduce, (4)
E+E	*i	reduce, (2)
E	*i	shift
E*	i	shift
E*i		reduce, (4)
E*E		reduce, (3)
E		reduce, (1)
S		ACCEPT

→ **shift-reduce conflict**.

Also: **reduce-reduce conflict**: letters on top of stack correspond to more than one production rule.

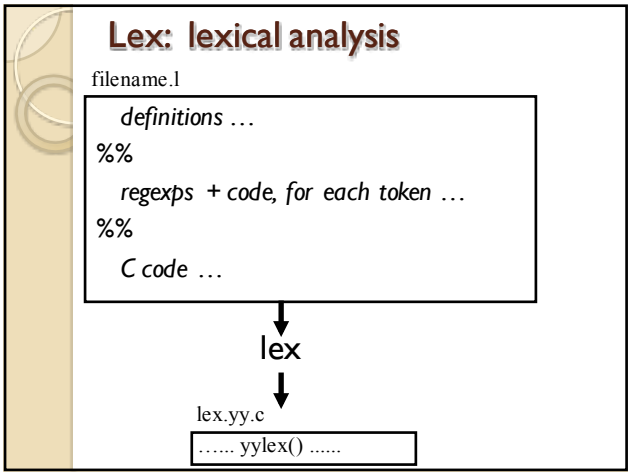
Yacc

- **Y**et **A**nother **C**ompiler-**C**ompiler
- A **parser-generator**
- Input: a Context-Free Grammar …
- Output: parser, in file `y.tab.c`
- Typically used with a lexical analyser, e.g., **Lex**.

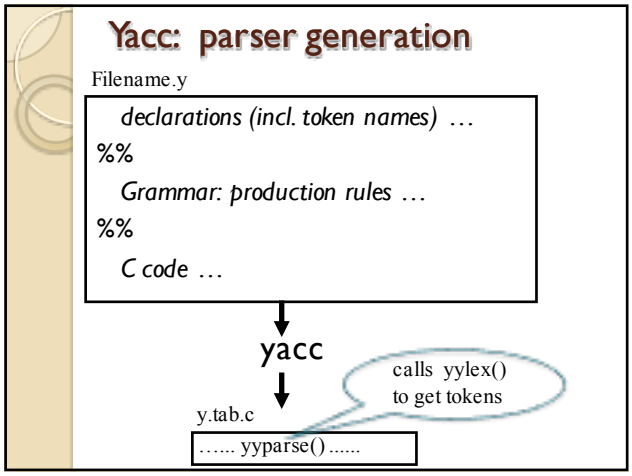
Lex

- **Lex**ical Analyser
- Input: regular expression for each token …
- Output: lexical analyser, in file `lex.yy.c`
- Both are widely available in Unix/Linux

Lex: lexical analysis



Yacc: parser generation



Lex & Yacc

- Compile `y.tab.c` and `lex.yy.c` using, say, `cc`.
- Obtain an executable parser.
- It can evaluate as it parses.
- See Assignment 2.
- Conflict resolution in Yacc:
  - Shift-reduce: shift
  - Reduce-reduce: use the rule listed first.

Revision

- Construct a parse tree for given string and grammar.
- Understand how a Shift-reduce Parser works.
- Start using Lex and Yacc.