CS344 - Parsing

Note Title 1/20/2012

Announcements

- First HW will be uploaded after class
  Essay

Why study programming languages?

- You will need to choose appropriate languages at some point.
- Makes it easier to learn new ones.
- Learn obscure features. - interview prep
- Knowledge of actual implementation costs.

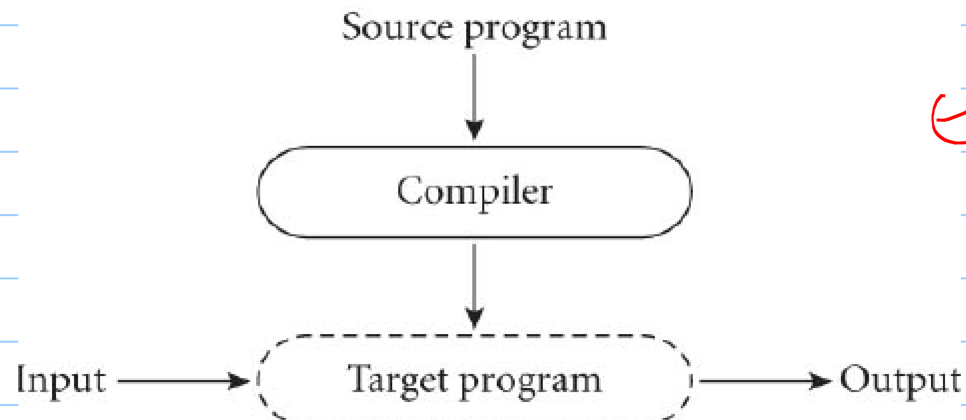
Ex: Housekeeping functions
Passing by reference

Why? (cont.)

- Make good use of debuggers, assemblers, etc.
- Add features to older languages as needed.

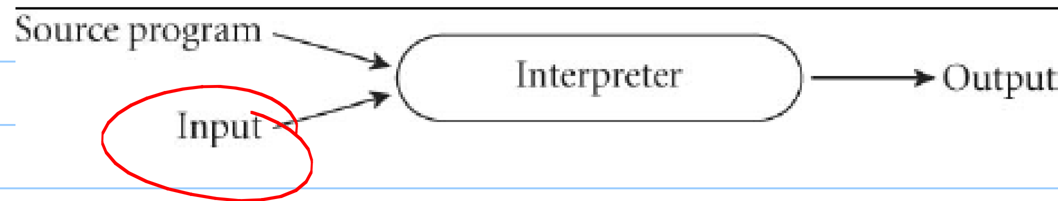
Compliation versus interpretation

2 models:



C++

Python



Pros + Cons

Interpreter:
- greater flexibility
- better debugging
- better with data that is dependant on input

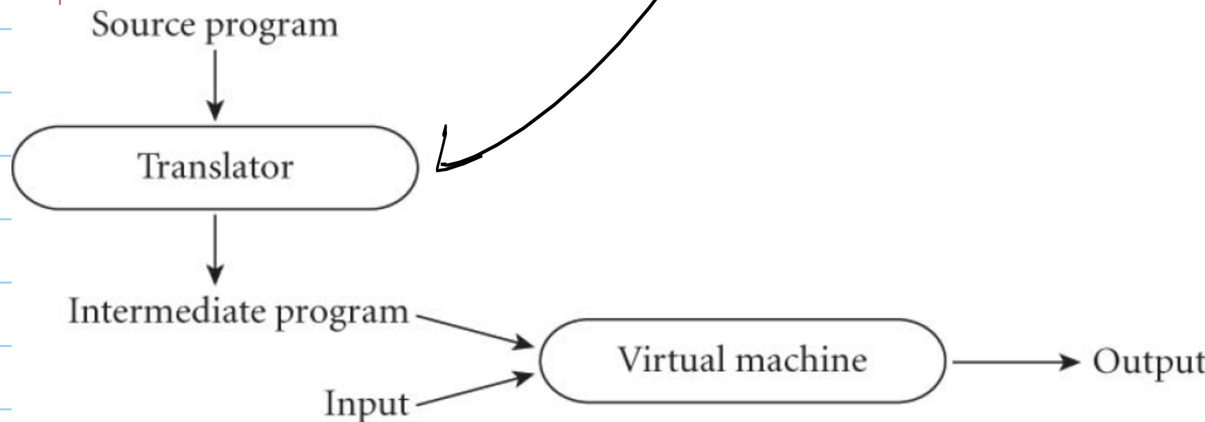
Compilation:
- *much* faster

Compilation vs. Interpretation

In reality, most languages are <u>both</u>.

This is the key.

Fuzzy: how much a translator does.



Compilers

The process by which programming languages are turned into assembly or machine code is important in programming languages.

We'll spend some time on these compilers athough it isn't a focus of this class.

Compilers

Compilers are essentially translaters, so must semantically understand the code

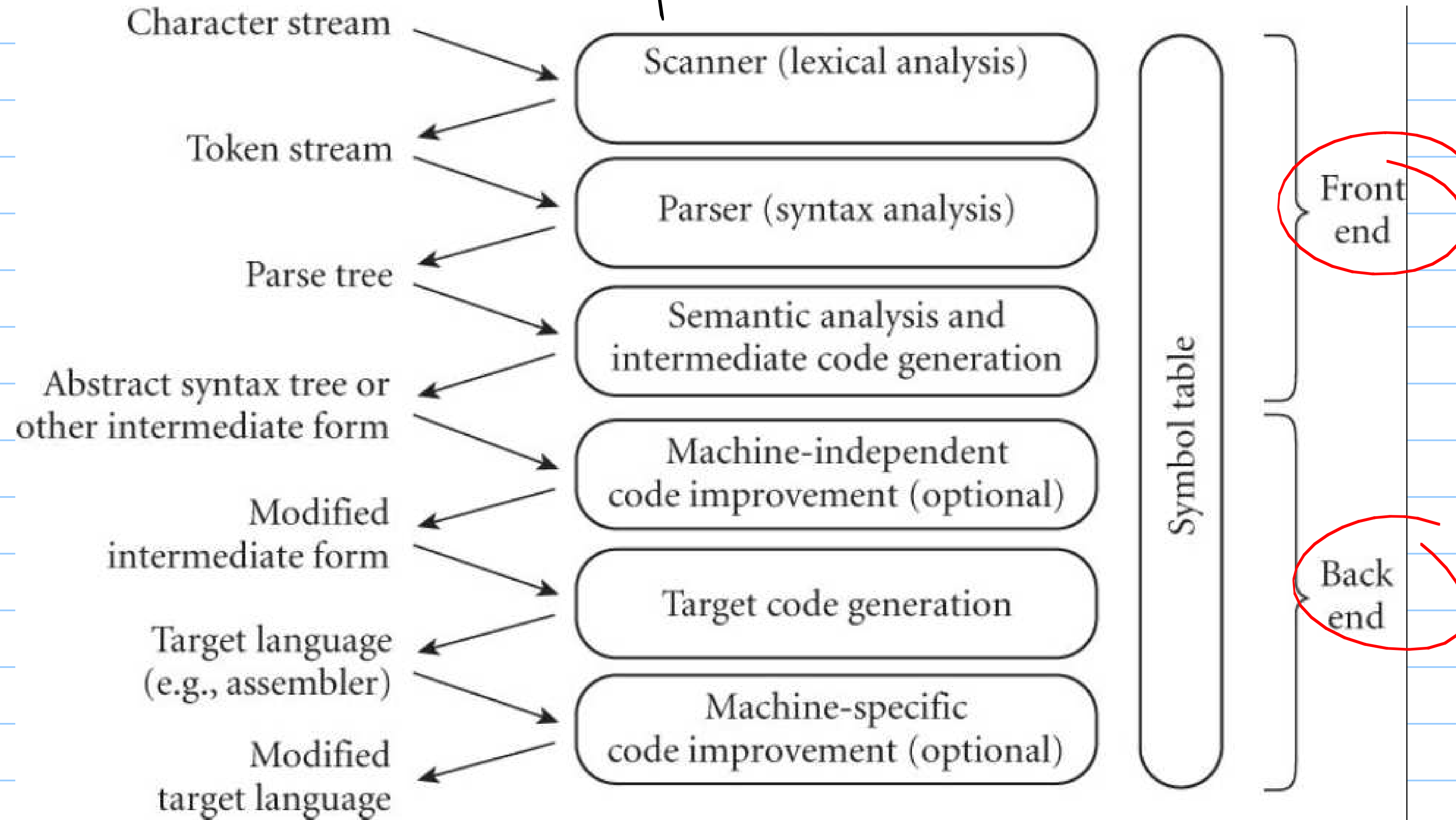
Output: either assembly, machine code or some other output

Java: byte code
C++ → Code

Compilers begin by preprocessing:

- remove whitespace + comments
- include macros or libraries
- group characters into tokens
  - ex: for (int i=0; i<10; i++)
    - i=i+2;
- identify high-level syntactical structures
  - ex: loops, functions

Overview of Compilation



Scanning (lexical analysis)

- Divide program into tokens, or smallest meaningful units

  Ex: keywords, ( ), { }, /n, ;

- Scanning & tokenizing makes parsing much simpler.

- While parsers can work character by character, it is slow.

- Note: Scanning is recognizing a regular language, eg via DFA

Parsing

compound_statement → { expresson }

- Recognizing a context-free language, e.g. via PDA
- Finds the structure of the program (or the syntax)

Ex: iteration-statement → while (expression) statement

statement → compound_statement

Outputs a parse tree.

Semantic Analysis

This discovers the *meaning* of the commands.

Actually only does *static* semantic analysis, consisting of all that is known at compile time.

(Some things - eg array out of bounds - are unknown until run time.)

Ex: (semantic analysis)

- Variables can't be used before being declared.
- Type checking.
- Identifiers are used in proper context.
- Functions have correct inputs & returns.

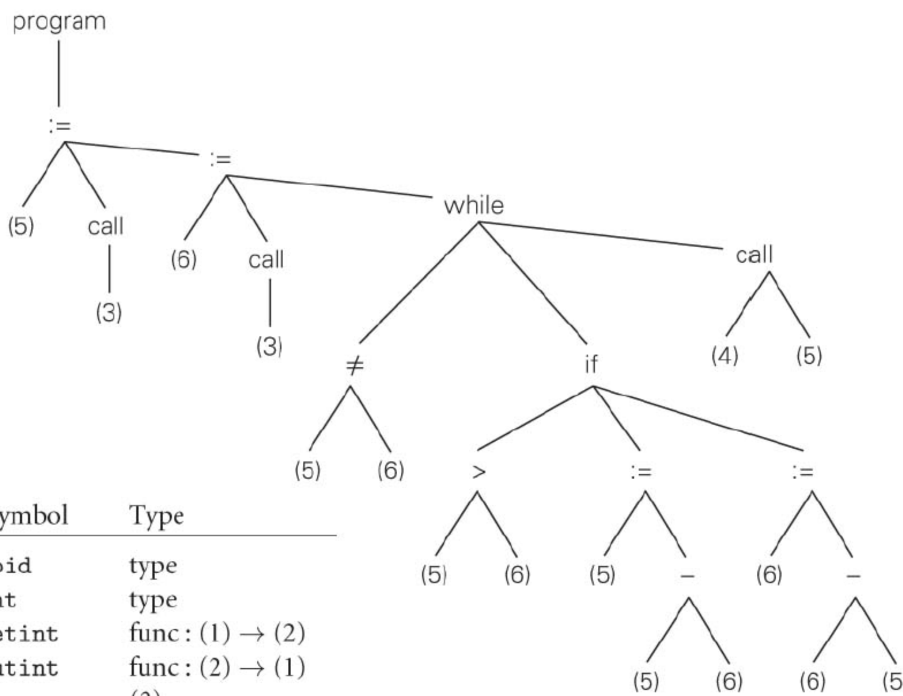
etc... (very language dependent)

Intermediate Form

This is the output of the "front end"

- Often, this is an abstract syntax tree — a simplified version of a parse tree

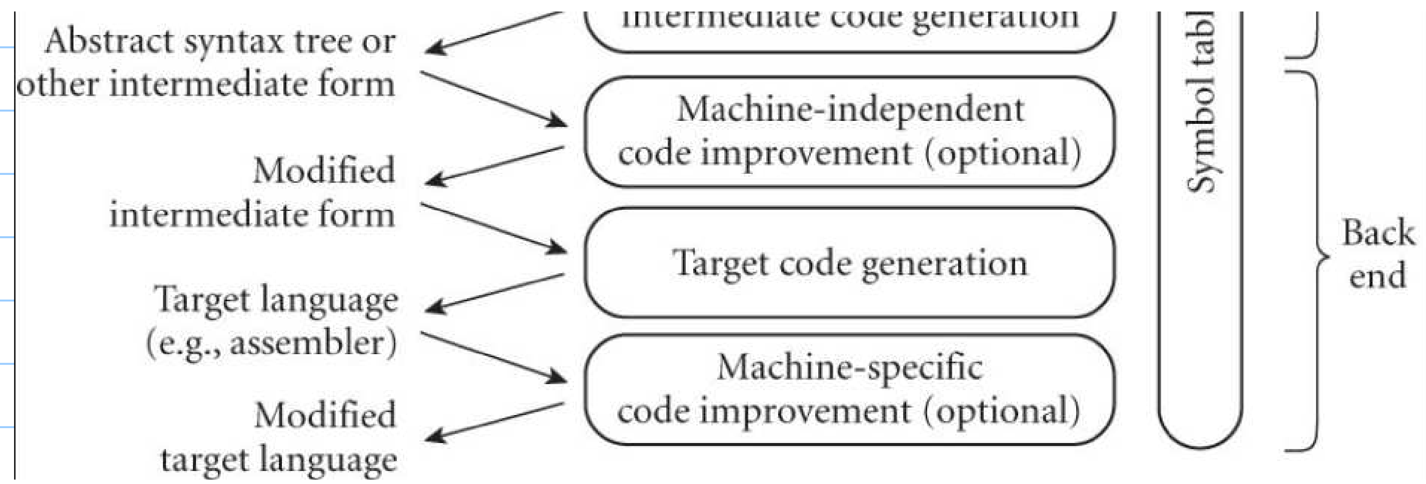
- May also be a type of assembly-like code



Index	Symbol	Type
1	void	type
2	int	type
3	getint	func : (1) → (2)
4	putint	func : (2) → (1)
5	i	(2)
6	j	(2)

$i = j + k;$

Code generation & improvement



Creating correct code is generally not difficult.

Optimization of that code is.

Next Time

Scanning and regular languages.