CS344 – Compilers: Scanning

Note Title 1/20/2012

Announcements

- We have moved!
  (Surprised me too...)
- Essay is due.
- Look for HW2 tomorrow.

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, although 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

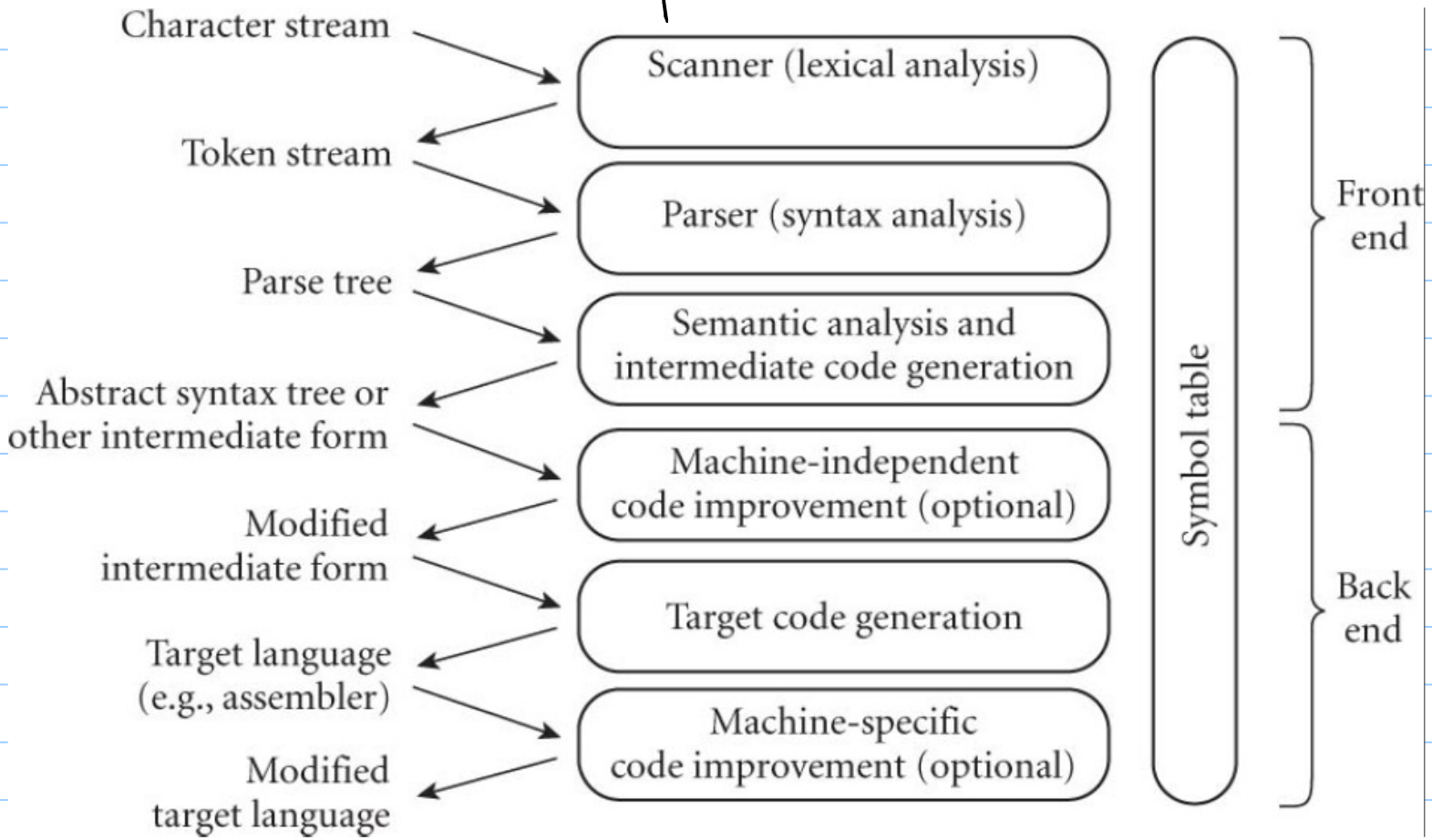
Java → bytecode

C++ → assembler

Compilers begin by preprocessing:

- remove white space ← comments
- include macros or libraries
- group characters into tokens
  ex: for (int i = 0; i < n; i++) {
- identify high-level syntactical structures
  ex:

Overview of Compilation



The steps:

Front end:

(A) Scanner

(B) Parser

(C) Semantic Analysis

Let's dive into these first...

Scanning (lexical analysis)

- Divide program into tokens, or smallest meaningful units

  Ex: keywords, {, }, +, variables, etc.

- 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

- 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 (after parsing)

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 unknowable until run time.)

→ error generation

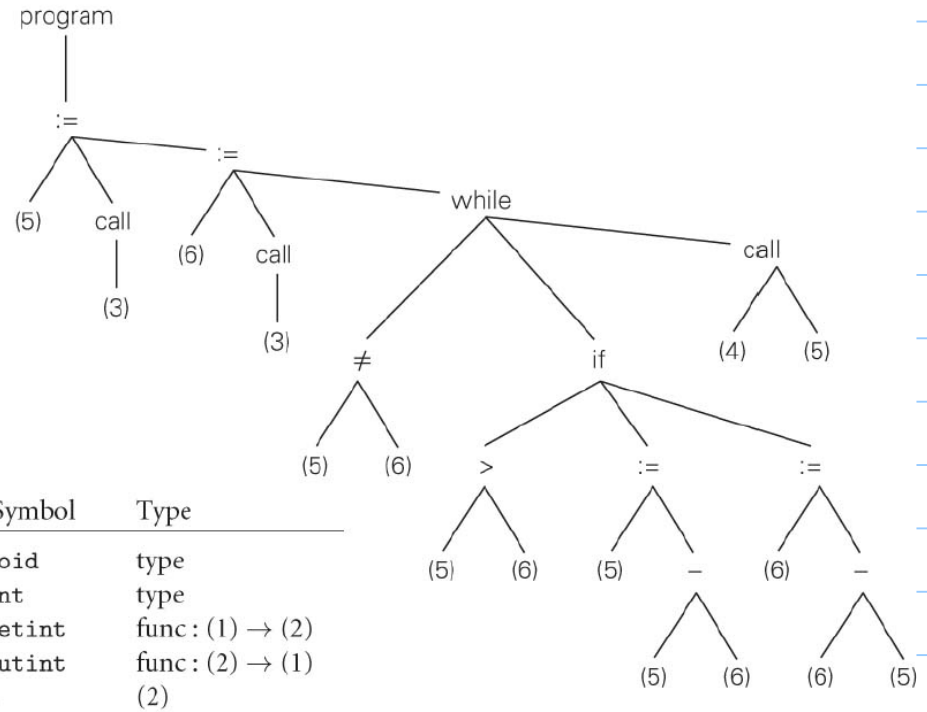
Ex: (semantic analysis)

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

etc... (very language dependent)

Intermediate Form

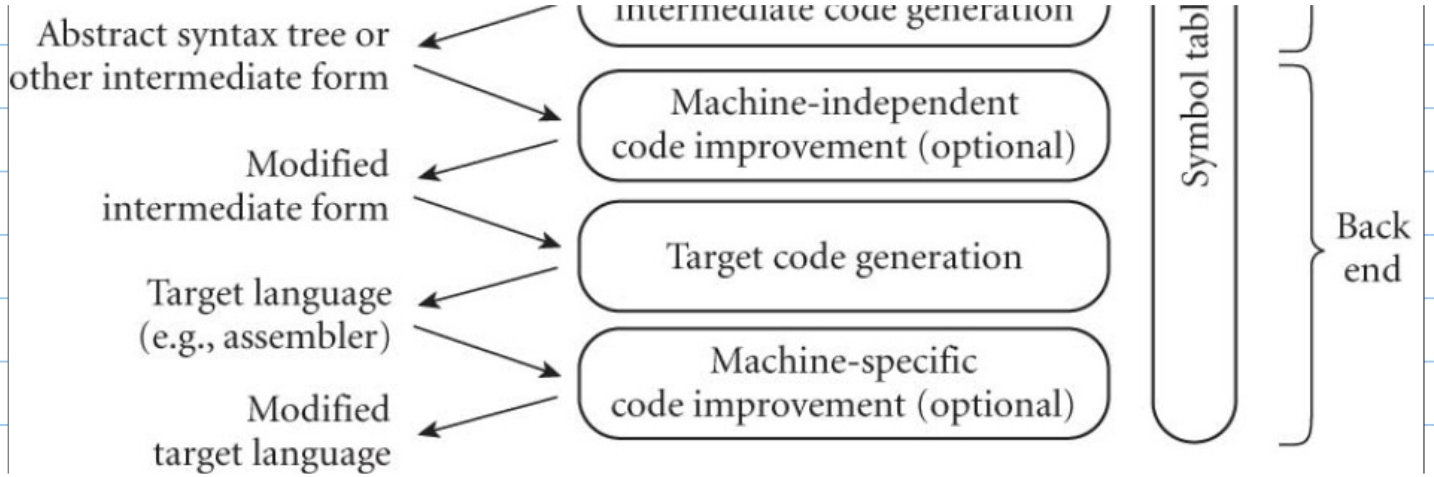
This is the output of the "front end"



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)

- Often, this is an abstract syntax tree — a simplified version of a parse tree
- May also be a type of assembly-like code

Back end: (Actual code generation)



Creating correct code is generally not difficult.

Optimization of that code is.

Back to front end:

(A) How is this actually done?

Input is actually a String of ASCII.

Need to find a way to scan letter by letter + decide what is a token.

Then pass the tokens on to the parser.

Regular Expressions: Some Theory

Defined as: Language generated as follows:

- A character: 0 or 1
- The empty string, ε
- 2 regular expressions concatenated
- 2 regular expressions separated by an or (written |)
- A regular expression followed by * (Kleene star – 0 or more ocurrances)

Regular Languages

The class of languages described by a regular expression.

Ex: $0^*10^* = L$

Any number of 0's, followed by a single 1, followed by any # of 0's.

$1 \in L$ $\quad 0 \notin L$

$001 \in L$ $\quad \epsilon \notin L$

Ex: Give the regular expression for
$\{w \mid w$ begins with a 1 and ends with a 0$\}$

$1(1 \mid 0)^* 0$

$(1 \mid 0)(1 \mid 0)(1 \mid 0)$ ← any # of times

Ex: $\{w \mid w$ starts with 0 and has an odd length$\}$

$= 0(11 \mid 10 \mid 01 \mid 00)^*$

$= 0((1 \mid 0)(1 \mid 0))^*$

Example: Numbers in Pascal

digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

unsigned_int → digit digit*

unsigned_number →
unsigned_int (ε | . unsigned_int)
(ε | ((e | E) (+ | - | ε) unsigned_int))

digit → [0-9]

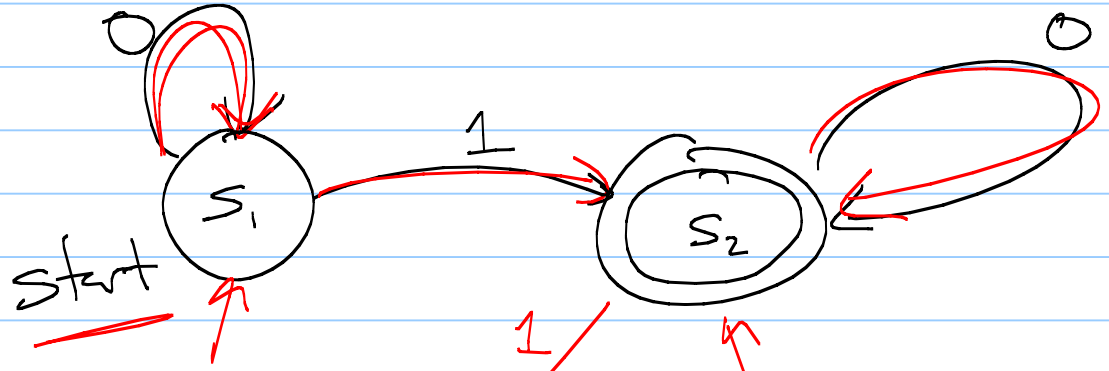
unsigned_num.
→ unsigned_int ε ε
→ digit digit* ε ε
→ 1

Deterministic Finite Automata (DFA)

Regular languages are precisely the things recognized by DFAs.

- A set of states
- input alphabet
- A start state
- A set of accept states
- A transition function: given a state & an input, output a new state

Example: word: 0010 $\in L$ recognized by DFA



$= 0^*10^*$

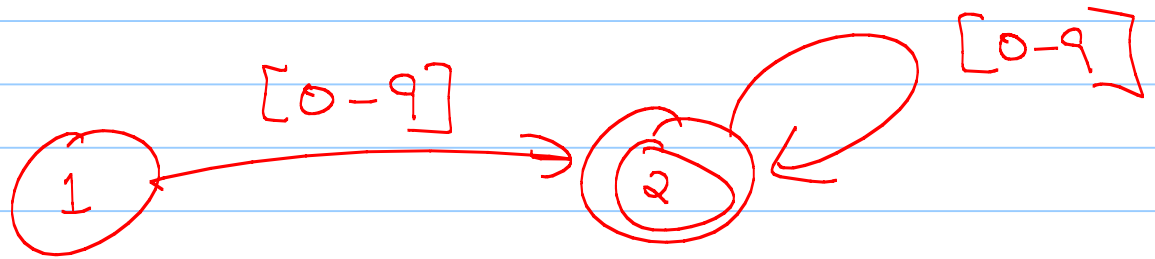
double circle indicates accept state

0|1

$00 \notin L$

Ex: unsigned_int → digit digit*

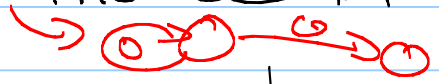
digit → [0-9]



DFA vs NFA

Non-deterministic Finite Automata

Note: No ambiguity is allowed in DFA's.

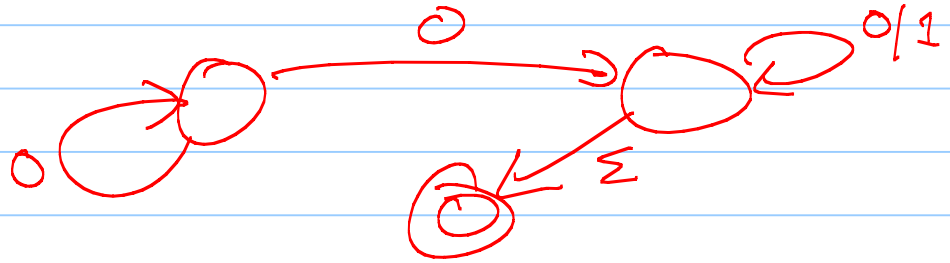


So given a state & input, can't be multiple options.

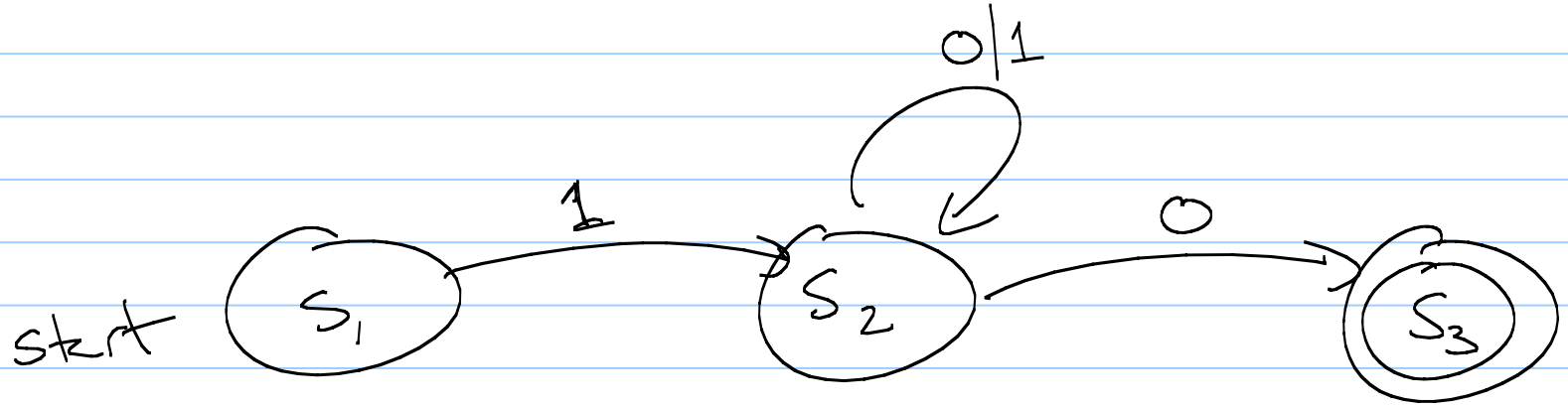
Also — no ε-transitions. (in DFA)

If we allow several choices to exist, this is called an NFA.

Ex:



Ex: $L = 1(0|1)^* 0$

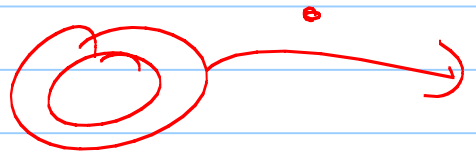


Ex: Some things are easier with NFA!

unsigned_number → unsigned_int (ε | . unsigned_int)

unsigned_int → [0-9]

DFA:



Essentially, we can think of an NFA as modeling a parallel set of possibilities (or a tree of them).

Thm: Every NFA has an equivalent DFA.
(size of DFA is much bigger ~ 2^n)

So: Both recognize reg. languages!

Limitations of Regular Expressions

Certain languages are not regular.

Ex: $\{w \mid w$ has an equal number of 0's and 1's$\}$

Somehow, this needs a type of memory, which regular expressions do not have.

(Pumping lemma)

Ex: $0^n 1^n$

reg? $0 0^* 1 1^*$

Why do we need this?

Need to "nest" expressions.

Ex: expr → id | number | -expr | (expr) | expr op expr

variable

non-terminal

op → + | - | / | *

Regular expressions can't quite do this.

Context Free Languages

Described in terms of productions
(Called Backus-Naur Form, or BNF)

- A set of terminals T : id, 6, +, ...
- A set of non-terminals N N → ____
- A start symbol S (a non-terminal)
- A set of productions

Ex: $\{ 0^n 1^n \mid n > 0 \}$

$$\begin{cases} S \to 0T1 \\ T \to 0T1 \mid \varepsilon \end{cases}$$

$$S \to 0S1 \mid 01$$

$n > 1$

Ex: $\{w \mid w$ has an equal number of 0s & 1s$\}$

Expression grammars: Simple calculator

expr → term | expr add_op term

term → factor | term mult_op factor

factor → id | numbr | -factor | (expr)

add_op → + | -

mult_op → * | /

Example: Show how rules can generate 3 + 4 * 5

Parse Tree

Ex: 3 + 4 * 5