CS344 - Compilers: Scanning

Announcements

- First HW due next Wed. (an essay)

- No class next Wednesday

- Make sure book is in by end of next week (ish)

- Next HW won't need book - will be up by Monday
Compilers

The process by which programming languages are translated 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 U class.
Compilers

Compilers are essentially translators, so must semantically understand the code.

Output: either assembly, machine code or some other output.

Java → byte code
Compilers begin by preprocessing:
- remove white space and comments
- include macros or libraries
- group characters into tokens
  ex: \[\text{for (int } i = 0; i < x; i++)}\]
- identify high-level syntactical structures
  ex: \[\text{if ( ) } i = i + 6 \text{ } \text{ cont } < \text{ } j\]
Overview of Compilation

Character stream

Token stream

Parse tree

Abstract syntax tree or other intermediate form

Modified intermediate form

Target language (e.g., assembler)

Modified target language

Scanner (lexical analysis)

Parser (syntax analysis)

Semantic analysis and intermediate code generation

Machine-independent code improvement (optional)

Target code generation

Machine-specific code improvement (optional)

Ch 2 of book
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: for, recognize keywords, group operations, names, etc.
- Scanning or tokenizing makes parsing much simpler.
- While parsers can work character by character, it is slow.
- Note: Scanning is recognizing a regular language, e.g., 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 - e.g. array out of bounds - are unknown until run time.)
By: (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
Back end: (Actual code generation)

Creating correct code is generally not difficult.
Optimization of that code is.
Back to front end:

How is this actually done?

Input is actually a string of ASCII.

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

Then pass the tokens on to the parsers.
Regular Expressions: some theory

A regular expression is defined (recursively) as:

- A character
- 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 occurrences)
Regular Languages
The class of languages described by a regular expression.

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

The set of 0-1 strings which contain exactly one 1.

$1 \in L$
$01 \in L$
$0100 \in L$
$11 \notin L$
$0 \notin L$
Exercise: Give the regular expression for \( \exists w \mid w \) begins with a 1 and ends 0 with a 0 of

\[ 1(0|1)^*0 \]

Exercise: \( \exists w \mid w \) starts with 0 and has an odd length

\[ 0((0|1)(0|1))^* \]
Example: Numbers in Pascal

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

unsigned_int → digit digit#

unsigned_number → unsigned_int unsigned_int

(unsigned_int (xₐ | (e | E) (q | -)(q | (unsigned_int))))
Deterministic Finite Automata (DFA)

Regular languages are precisely the 0 things recognized by DFAs.

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

\[ S_0 \xrightarrow{0} S_1 \xrightarrow{1} S_2 \xrightarrow{0} S_0 \]

\[ S_3 \in \{0, 1\} \]

Inputs: 0, 1

Transitions - indicated with arrows

0 1 0

\( S_0 \) is the accept state
Ex: `unsigned_int` → `digit digit*`
`digit` → `[0-9]`

DFA:
Non-deterministic Finite Automata: NFA

Note: No ambiguity is allowed in DFA's. Given a state and input, can't be multiple options.

Also—no ε-transitions.

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

Ex:

```
string 11
has 2 possible paths in NFA
```
$L = 1 (0|1)^* 0$

Diagram:

- Start state: $S_1$
- State $S_2$
- State $S_3$
- Transitions:
  - From $S_1$ to $S_2$ on input $1$
  - From $S_2$ to $S_1$ on input $0|1$
  - From $S_2$ to $S_3$ on input $0$

Input: $10001L$
Ex: Some things are easier with NFA!

unsigned_number → unsigned_int (3 | . unsigned_int)
unsigned_int → [0-9]

NFA:
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.

So: Both recognize regular languages!
Limitations of Regular Expressions

Certain languages are not regular.

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

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

$0^n 1^n$
Why do we care?

Need to "nest" expressions.

\[ \text{Ex: } \text{expr} \rightarrow \text{id | number | -expr | (expr) | expr op expr} \]
\[ \text{op} \rightarrow + | - | \mid | \ast \]

Regular expressions can't quite do this.

(This will come up more in parsing - next week or later)
Scannness: do this in code

Find the syntax (not semantics) of code.

Output tokens.

A few types:

- Ad-hoc

- Finite automata
  - nested case statements
  - table + driver
Ad-hoc: case based code

if \( \text{current} \in \{ \text{"(", ",", "}\} \)
    \( \text{return that symbol} \)
if \( \text{current} = \text{";"} \)
    \( \text{read next} \)
    if \( \text{it is = } \) announce "assign"
        else announce error
if \( \text{current} = \text{"/"} \)
    \( \text{read next} \)
    if \( \text{it is } \text{"*/" or "newline" (resp.)} \)
        \( \text{read until } \text{"*/" or "newline" (resp.)} \)
        \( \text{else return divide} \)
else return divide

\( \text{etc.} \)
Ad-hoc approach

Advantage:
- Code is fast and compact

Disadvantage:
- Very ad-hoc!
- Hard to debug
- No explicit representation
DFA approach:

Given a regular expression, convert to a DFA.

We'll walk through this next week—in Ch 2 of the book.

However
Scanning Programs

In reality, this DFA is often done automatically. Specify the rules of regular language, the program does this for you.

Many such examples:

Lex (flex), JLex/JFlex, Quex, Ragel, ...
Next time:

Lex / Flex: C-style driver

Look for HW on regular expressions, NFA/PFA, and context-free languages.

Next programming assignment will use Flex.