CSE 401 – Compilers

Lecture 2: Languages, Automata, Regular Expressions & Scanners

Michael Ringenburg

Winter 2013

Administrative Notes

- Reading
  - Cooper & Torczon: Chapter 1, and Sections 2.1-2.4
  - Try to finish by the end of the week – it'll be helpful for the first homework.
- First homework
  - Should be out on Friday (I'll post on course website and send an email).
  - Will be due a week from Friday (January 18).
  - Note: You have 4 late days *for the entire quarter*. Use them wisely (see syllabus for details).

Reminders

- Please vote for office hours by end-of-day Thursday (see link on course home page).
  - Select whichever slots you think you could reasonably attend.
  - We will use this to help decide office hours for the TAs and the instructor.
- Please pick your project partner, and send mail to cse401-staff[at]cs.
  - First piece of the project will be released (early) next week, so you should pick partners *this week*.

Snow

- It's the time of year where the "S"-word starts to show up occasionally in weather forecasts.
- The schedule for the quarter is tight, so if we do have a snow day at some point, we may have to rush through some of the material.
  – If this happens, take advantage of the extra time on the snow day to pay extra attention to the readings – with less time to cover the material in class, the readings become correspondingly more important.

Agenda

- **Finish course intro (history)**
- Introduce Scanning (part 1 of your project)
  - Quick review of basic concepts of formal grammars
  - Regular expressions
  - Lexical specification of programming languages
  - Using finite automata to recognize regular expressions
  - Scanners and Tokens

Some History

- Early computers – hand coded assembly language (punchcards!)
  - Hard to write anything complex – but earliest computers couldn't execute any thing that comlex.
- 1952: Grace Hopper writes first compiler (for A-0), and coins the term "compiler".
  - Essentially a collection of mathematical subroutines that could be called. The compiler would take a series of calls and convert them into an executable.
  - Successors: A-1, A-2 (first "open source" software), and later … B-0!
- 1957: IBM writes first real "high-level" language compiler, for FORTRAN. (FORTRAN *is* high level compared to assembly.)
  - Competitive with hand-optimized code.
  - Required 18 person-years (hopefully your projects won't take this long!)

Some History

- 1962: First *bootstrapped* compiler (for LISP)
  - A compiler that was compiled by itself, rather than written in assembly (or another language).
  - Requires initially creating a very simple compiler in assembly or another language, and then using that to compile the initial bootstrapped compiler.
    - Initial compiler may contain just a subset of the language. As this compiler is refined to compile more of the language, the compiler itself can begin to use more of the language.
  - Much more efficient that writing in assembly (like the first compilers).
  - Great way to test a compiler.
- Rest of 1960's, into 1970's
  - Work on formalizing scanning and parsing (theory and practice).
  - Automatic parser and scanner generators
    - Lex (lexical analyzer) and Yacc (Yet Another Compiler Compiler)
    - JFlex and Cup are direct descendants of these C-based tools.

Some History

- Late 1970's, 1980's
  - New languages (functional; object-oriented)
  - New architectures (RISC, parallel machines, caches, …)
  - Back-end improvements: Optimization, Register Allocation, Automatic parallelization
- 1990s
  - Improved techniques for compiling object oriented code
    - Efficiency in the presence of dynamic dispatch and small methods
  - Just-in-time compilers (JITs)
  - Compiler technology to effectively use new hardware (RISC, parallel machines, complex memory hierarchies)

Some History

- Last decade
  - Compilation techniques in many new places
    - E.g., parsing, semantic analysis, source-to-source translation used for software analysis, verification, security
  - Phased compilation – blurring the lines between "compile time" and "runtime"
    - Programs can generate and compile specialized versions of routines "on the fly".
    - Can use machine learning to control optimizations
  - Multicore: parallelism everywhere!

Any questions?

- Don't hesitate to ask – I'm teaching this course because I enjoy talking about compilers.
- If you have a question, it's likely other people do as well, but they are too shy to ask. So you'll be doing them a favor too.

Agenda

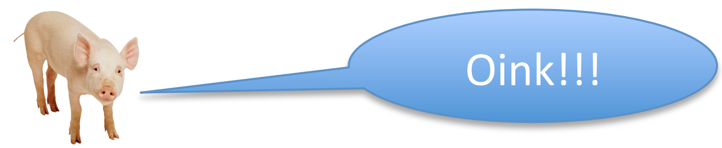
- Finish course intro
- **Introduce Scanning (part 1 of your project)**
  - Quick review of basic concepts of formal grammars
  - Regular expressions
  - Lexical specification of programming languages
  - Using finite automata to recognize regular expressions
  - Scanners and Tokens

Programming Language Specifications

- Since the 1960s, the syntax of every significant programming language has been specified by a formal grammar
  - If you ever have the "pleasure" of reading a language specification document, you'll see that each section typically consists of a formal grammar for some piece of the syntax, followed by notes describing the semantics.
  - First done in 1959 with BNF (Backus-Naur Form) grammar used to specify ALGOL 60 syntax
  - Borrowed from the linguistics community (Chomsky)

Review of Formal Languages and Automata Theory

- Starring Mr. Pig



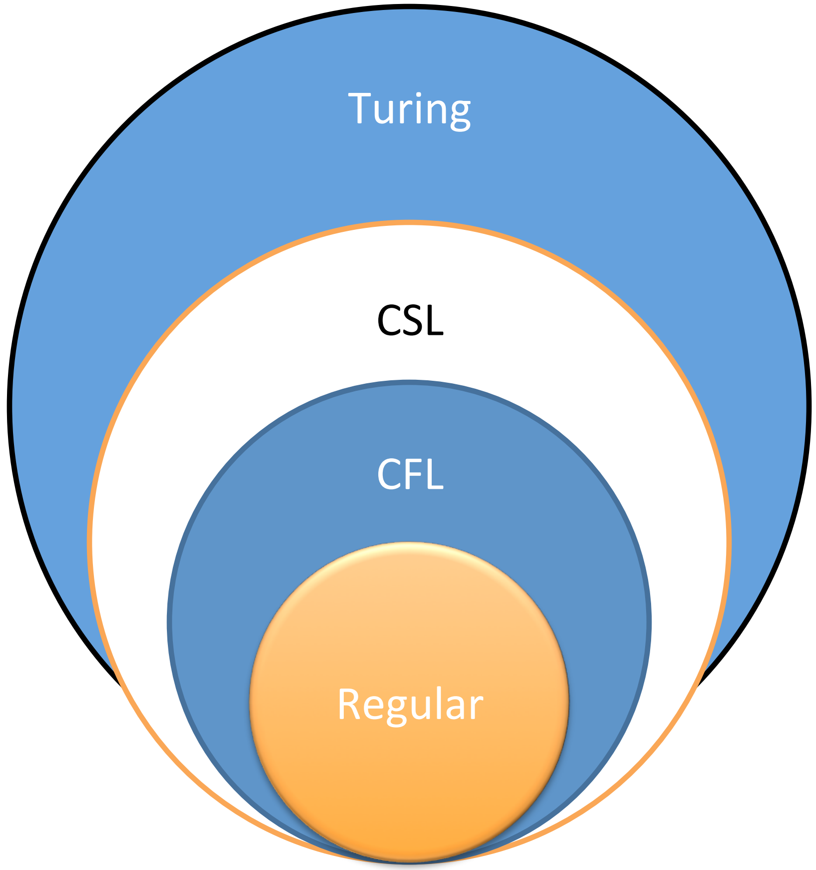
- Alphabet: a finite set of symbols and characters
  - E.g., {‘i’, ‘k’, ‘n’, ‘o’, ‘!’, ‘ ’}
- String: a finite, possibly empty sequence of symbols from an alphabet
  - E.g., “oink”
- Language: a set of strings (possibly empty or infinite)
  - E.g., {“oink!”, “oink oink!”, “oink oink oink!”, …}

Finite Specifications of Possibly Infinite Languages

- Automaton – a recognizer; a machine that accepts all strings in a language (and rejects all other strings)
  - E.g., a pig detector: accepts all sequences of oinks, rejects "moo"s or "baa"s
- Grammar – a generator that produced all strings in the language (and nothing else)
  - Unfortunately, we can't use a pig as our grammar – no pig (that I've met) can generate infinite "oink" sequences.
  - Instead we use formal (aka mathematical) grammars.
- A particular language may be specified by many different grammars and automata
  - *But*, a grammar or automaton specifies only one language

Language (Chomsky) hierarchy: quick reminder

- Regular (Type-3) languages are specified by regular expressions/ grammars and finite automata (FAs) ← SCANNING
- Context-free (Type-2) languages are specified by context-free grammars and pushdown automata (PDAs) ← PARSING
- Context-sensitive (Type-1) languages … aren't too important
- Recursively-enumerable (Type-0) languages are specified by general grammars and Turing machines



Example: Grammar for Pig-ish (or Pig-ese?)

- A formal grammar for our pig language could be:

PigTalk ::= oink *PigTalk*		(rule 1)
|	oink!	(rule 2)

- *PigTalk* can then generate, for example:

1) *PigTalk* **::=** oink!		(Rule 2)
2) *PigTalk* **::=** oink *PigTalk*		(Rule 1)
	::= oink oink!	(Rule 2)
3) *PigTalk* **::=** oink *PigTalk*		(Rule 1)
	::= oink oink *PigTalk*	(Rule 1)
	::= oink oink oink!	(Rule 2)

Example: Grammar for a Tiny Language

- A more realistic (but still small) language:

program ::= *statement* | *program statement*

statement ::= *assignStmt* | *ifStmt*

assignStmt ::= *id* = *expr* ;

ifStmt ::= if (*expr*) *statement*

expr ::= *id* | *int* | *expr* + *expr*

id ::= a | b | c | i | j | k | n | x | y | z

int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

More Formally

- The rules of a grammar are called *productions*
- Rules contain
  - Nonterminal symbols: grammar variables (*program, statement, id,* etc.)
  - Terminal symbols: concrete syntax that appears in programs (a, b, c, 0, 1, if, =, (,), …
- Meaning of

nonterminal ::= <sequence of terminals and nonterminals>

  - In a derivation, an instance of *nonterminal* can be replaced by the sequence of terminals and nonterminals on the right of the production
- Often there are several productions for a nonterminal – derivations can choose any of them.

Exercise 1: Derive a simple program

program ::= *statement* | *program statement*
statement ::= *assignStmt* | *ifStmt*
assignStmt ::= *id* = *expr* ;
ifStmt ::= if (*expr*) *statement*
expr ::= *id* | *int* | *expr* + *expr*
id ::= a | b | c | i | j | k | n | x | y | z
int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

program **::=**
statement **::=**
???

```
if (x) y = 1 + y ;
```

Exercise 1 (solution): Derive a simple program

program ::= *statement* | *program statement*
statement ::= *assignStmt* | *ifStmt*
assignStmt ::= *id* = *expr* ;
ifStmt ::= if (*expr*) *statement*
expr ::= *id* | *int* | *expr* + *expr*
id ::= a | b | c | i | j | k | n | x | y | z
int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

```
if (x) y = 1 + y ;
```

This is just one possible derivation.
Many others are possible.

program **::=**
statement **::=**
ifStmt **::=**
if (*expr*) *statement* **::=**
if (*id*) *statement* **::=**
if (x) *statement* **::=**
if (x) *assignStmt* **::=**
if (x) *id* = *expr* ; **::=**
if (x) y = *expr* ; **::=**
if (x) y = *expr* + *expr* ; **::=**
if (x) y = *int* + *expr* ; **::=**
if (x) y = 1 + *expr* ; **::=**
if (x) y = 1 + *id* ; **::=**
if (x) y = 1 + y ;

Exercise 2: A multistatement program

program ::= *statement* | *program statement*
statement ::= *assignStmt* | *ifStmt*
assignStmt ::= *id* = *expr* ;
ifStmt ::= if (*expr*) *statement*
expr ::= *id* | *int* | *expr* + *expr*
id ::= a | b | c | i | j | k | n | x | y | z
int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

program **::=**
???

```
if (x) y = 1 + y ; x = 1 ;
```

Your solution may reference your previous derivation.

Exercise 2 (solution): A multistatement program

program ::= *statement* | *program statement*
statement ::= *assignStmt* | *ifStmt*
assignStmt ::= *id* = *expr* ;
ifStmt ::= if (*expr*) *statement*
expr ::= *id* | *int* | *expr* + *expr*
id ::= a | b | c | i | j | k | n | x | y | z
int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

```
if (x) y = 1 + y ; x = 1 ;
```

program **::=**
program statement **::=**
program assignStmt **::=**
program id = *expr* ; **::=**
program x = *expr* ; **::=**
program x = *int* ; **::=**
program x = 1 ; **::=**

Then derive *program* as in the previous example.

Once again, others are possible.

Alternative Notations

- There are several syntax notations for productions in common use; all mean the same thing. E.g.:

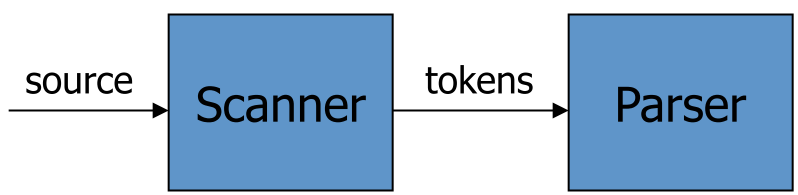
ifStmt ::= if (*expr*) *statement*
ifStmt ⟶ if (*expr*) *statement*
<ifStmt> ::= if (<expr>) <statement>

Parsing

- Parsing: reconstruct the derivation (syntactic structure) of a program
- In principle, a single recognizer could work directly from a concrete, character-by-character grammar
- In practice this is never done

Parsing & Scanning

- In real compilers the recognizer is split into two phases*
  - Scanner: translate source code to tokens (e.g., <int>,+,<id>)
    - Reports *lexical* errors like illegal characters and illegal symbols.
  - Parser: read token stream and reconstruct the derivation
    - Reports *parsing* errors – i.e., source that is not derivable from the grammar. E.g., mismatched parenthesis/braces, nonsensical statements (x = 1 +;)



*Not always quite this clean of a separation (as we'll see later) – but true at a high level.

Why Separate the Scanner and Parser?

- Standard arguments about splitting functionality into independent pieces: Simplicity & Separation of concerns
  - Scanner hides details from parser (comments, whitespace, input files, etc.)
  - Parser is easier to build; has simpler input stream (tokens) and narrow interface
- Efficiency
  - Tokens can be defined by regular expressions, and recognized by finite automata.
    - (But still often consumes a surprising amount of the compiler's total execution time) ← File I/O!
  - Parsing requires context-free grammars, and thus pushdown automata.
  - Can build automatic DFA generators for scanning (Jflex) and automatic PDA generators for parsing (CUP).

But …

- Not always possible to separate cleanly
- Example: C/C++/Java *type* vs *identifier*
  - Parser would like to know which names are types and which are identifiers, but
  - Scanner doesn't know how things are declared …
- Things are even uglier in Fortran 77
  - E.g., `myvar`, `my var`, and `my  var` are all the same identifier, keywords are not reserved, etc. Tokenizing requires context (see Cooper & Torczon 2.6 if you are curious).
- So we hack around it somehow…
  - Either use simpler grammar and disambiguate later, or communicate between scanner & parser (with some semantic analysis mixed in).
  - Real world: Often ends up very complex and hard to follow. Compiler front ends are sometimes referred to as "black magic".
  - Not for your project though – language is simplified.

Typical Tokens in Programming Languages

- Operators & Punctuation
  - `+ - * / ( ) { } [ ] ; : :: < <= == = != !` …
  - Each of these is a distinct lexical class
- Keywords
  - `if while for goto return switch void` …
  - Each of these is also a distinct lexical class (*not* a string)
- Identifiers (variables)
  - A single ID lexical class, but *parameterized by actual identifier* (often a pointer into a symbol table).
- Integer constants
  - A single INT lexical class, but *parameterized by numeric value*
- Other constants (string, floating point, boolean, …), etc.

Principle of Longest Match

- In most languages (exception: Fortran 77), the scanner should pick the longest possible string to make up the next token if there is a choice
- Example:

```
return maybe != iffy;
```

should be recognized as 5 tokens:

RETURN	ID(maybe)	NEQ	ID(iffy)	SCOLON

not 7:

RETURN	ID(maybe)	NOT	ASSIGN	IF	ID(fy)	SCOLON

Lexical Complications

- Most modern languages are free-form
  - Layout doesn't matter
  - Whitespace separates tokens
- Alternatives
  - Haskell, Python – indentation and layout can imply grouping
- And other confusions
  - In C++ or Java, is >> a shift operator or the end of two nested templates or generic classes?

Regular Expressions and Finite Automate (FAs)

- The lexical grammar (structure) of most programming languages can be specified with regular expressions
  – (Sometimes a little cheating is needed)
- Therefore, tokens can be recognized by a deterministic finite automaton
  – Can be either table-driven or built by hand based on lexical grammar

Regular Expressions

- Defined over some alphabet Σ
  - For programming languages, alphabet is usually ASCII or Unicode
- If *re* is a regular expression, *L*(*re*) is the language (set of strings) generated by *re*

Fundamental REs

re	*L*(*re*)	Notes
a	{ a }	Singleton set, for each symbol a in the alphabet Σ
ε	{ ε }	Empty string
∅	{ }	Empty language

These are the basic building blocks that other regular expressions are built from.

Operations on REs

re	*L*(*re*)	Notes
rs	L(r)L(s)	Concatenation – r followed by s
r|s	L(r) ∪ L(s)	Combination (union) – r or s
r*	L(r)*	0 or more occurrences of r (Kleene closure)

Precedence: * (highest), concatenation, | (lowest)

Parentheses can be used to group REs as needed

Next time

- We'll continue discussing Regular Expressions
- We'll also discuss how to build finite automata that recognize Regular Expressions, and show how they are used to build scanners.