Lexical Analysis

David Notkin
Autumn Quarter 2008

Survey (partial results)

Excited [selected responses]

- My dad says it's the epitome of Computer Science.
- I want to understand how a compiler works. See behind the scenes and learn something about software engineering.
- I really liked 341, and this seems like a way to understand more of what goes on behind programming languages.
- I am excited about the project and actually implementing some of the theoretical knowledge.
- Haven't heard much, but I'm thinking it will have to do a lot with the stuff we learned in 322 such as grammars and regular expressions.
- Sadly, I haven't heard much, but I enjoyed the first lecture.

Concerned [selected responses]

- Very difficult projects
- High work load and difficult project
- Some think it's a bunch of pointless theory.
- I am worried about getting too buried in theory that isn't shown in an applicable way. Quite a bit of work and complicated.
- The projects are a lot of work, especially when you are taking 3 other courses (2 of which are CSE
- I've heard the projects can be time-consuming
- I've heard that most of the work is figuring what the preexisting project code does than applying the theory we learn. While that is a useful skill, I hope to get more out of the projects.

Survey (partial results): interests…

- I was interested in how compilers are able to take the Fibonacci function recursively written and optimize it into a for loop without the user knowing it. I see now optimizations aren't part of the class, but I would like to learn maybe about just what optimizations are used in compilers today.
- No specific ones yet. I suppose I'm a little curious about how ML (and similar statically typed languages) do type inference.
- I am especially interested in JIT compiling.
- Effectively parsing text is something I've been curious about, so i'm looking forward to that. Also getting to the computer to recognize the meaning of text looks interesting.
- What is being done in the field of compilers that try to make code more secure?
- I'm interested in seeing how theory connects to practice.
- I'm not sure if this relates to this course, but I'm interested in learning how scripting languages are compiled since we don't go through a build process with them as we do with Java or C, if it's a different process.

Question

- **IF** I were to offer the following ***option***, might you be interested in it?
  - A reduced project (that is, not all of the extensions to MiniJava but still enough to learn about key issues in compiling) ***and***
  - more substantial "homework"
- If you are interested in this option, send email either directly to me or, if you prefer, to the mailing list **before lecture** on Monday
- I will decide if this is feasible by Wednesday's lecture
  - It's not a vote, it's my decision: articulate arguments, in addition to degree of interest, will help inform my decision

Scanning a.k.a. lexing: purpose

- Turn the character stream that represents the source program into a token stream
  - In general, it should be an efficient phase of compilation
- A token is a group of characters forming an atomic unit of syntax, such as a identifier, number, etc.
- White space comprises
  - characters between tokens that are ignored
  - they contribute to the human communication aspects of the program, but do not change the semantics of its execution

Separating lexing from parsing

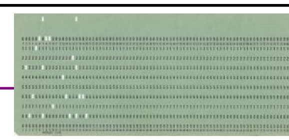
- Lexing can be represented and implemented as part of syntactic analysis
  - Regular expressions are a proper subset of context free grammars
- But this is rarely if ever done: separating concerns tends to be a better design
- Simplifies scanner and parser
  - Scanner handles I/O and machine dependencies, needn't know language syntax
  - Recognizing regular expressions is much faster than parsing context free grammars
  - Parser can focus on syntactic structure

Lexical design

- Most languages are free form
  - Layout doesn't matter (to the computer – see obfuscated code example on right)
  - White space separates tokens
- But some languages are more constrained lexically

```
#include <stdio.h>
main(t,_,a)char *a;{return!0<t?t<3?main(-79,-
13,a+main(-87,1-_,
main(-86,0,a+1)+a)):1,t<_?main(t+1,_,a):3,main(-
94,-27+t,a)&&t==2?_<13?
main(2,_+1,"%s %d %d\n"):9:16:t<0?t<-
72?main(_,t,
"@n'+,#'/*{}w+/w#cdnr/+,{}r/*de}+,/*{*+,/w{%+,/w
#q#n+,/#{l,+,/n{n+,/+#n+,/#\
;#q#n+,/+k#;*+,/'r :'d*'3,}{w+K w'K:'+}e#';dq#'l
\
q#'+d'K#!/+k#;q#'r}eKK#}w'r}eKK{nl]'/#;#q#n'){)#
}w'){){nl]'/+#n';d}rw' i;# \
){nl]!/n{n#'; r{#w'r nc{nl]'/#{l,+'K {rw'
iK{;[{nl]'/w#q#n'wk nw' \
iwk{KK{nl]!/w{%'l##w#' i;
:{nl]'/*{q#'ld;r'}{nlwb!/*de}'c \
;;{nl'-
{}rw]'/+,}##'*}#nc,',#nw]'/+kd'+e}+;#'rdq#w!
nr'/ ') }+}{rl#'{n' ')# \
}'+}##(!!/")
:t<-50?_==*a?putchar(31[a]):main(-
65,_,a+1):main((*a=='/')+t,_,a+1)
  :0<t?main(2,2,"%s"):*a=='/'||main(0,main(-
61,*a,
"!ek;dc i@bK'(q)-[w]*%n+r3#l,{}:\nuwloca-O;m
.vpbks,fxntdCeghiry"),a+1);}
```


Ex: Fortran



- Data cards
- Comment cards: first character is 'C'
- Statement cards
  - First five characters are an optional statement number
  - Sixth character is a continuation character – any character other than '0' indicates that this continues the statement from the previous card
  - Characters 7 through 72 are source code
  - Characters 73 through 80 are optional and have no meaning with respect to the program *per se*

Ex: Haskell

- "[I]ndentation … is important. Haskell uses a system called 'layout' to structure its code (… Python uses a similar system). The layout system allows you to write code without the explicit semicolons and braces that other languages like C and Java require."
  -- Hal Daumé III
- Tabs and spaces can cause confusion

```
main = let dolly = breedSheep
       in do args <- getArgs
             print $ traceFamily dolly (map getFunctionByName args)
```


Definitions

- Pattern: a definition of a related set of lexical entities
  – Ex: all sequences of numeric characters, all sequences of alphanumeric characters starting with an alphabetic character
  – Regular expressions are used in practice to define patterns
- Lexeme: group of characters that matches a pattern
  – Ex: '1234', '43204222', 'snork', 'f0rk'
- Token: class of lexemes matching a pattern, distinguished by an attribute
  – Ex: 'snork' and 'f0rk' are both identifier lexemes with the actual names kept as an attribute

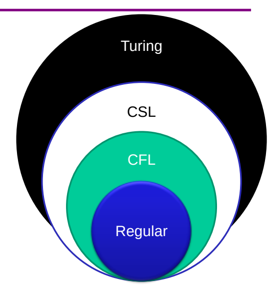
Languages: quick reminder

- **Alphabet**: finite set of characters and symbols
- **String**: a finite (possibly empty) sequence of characters from an alphabet
- **Language**: a (possibly empty or infinite) set of strings
- **Grammar**: a finite specification for a set of strings
- **Language automaton**: an abstract machine that accepts all strings in a given language and only those

- A language can be specified by many different grammars and automata
- A grammar or automaton specifies a single language

Language (Chomsky) hierarchy: quick reminder

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



One distinction among the levels is what is allowed on the right hand and on the left hand sides of grammar rules

Regular Expressions: defined inductively

- Base cases
  - Empty string (ε)
  - Symbol from the alphabet
- Inductive cases
  - Concatenation: E_1E_2
  - Alternation $E_1 \mid E_2$
  - Kleene closure: E*
- Parentheses for grouping
- Precedence: * is highest, then concatenate, | is lowest
- White space not significant

Notational Conveniences: no additional expressive power

- E+ ≅ 1 or more occurrences of E
- Ek ≅ exactly k occurrences of E
- [E] ≅ 0 or 1 occurrences of E
- {E} ≅ E*
- not(x) ≅ any character in alphabet except x
- not(E) ≅ any strings from alphabet except those in E
- E1-E2 ≅ any string matching E1 that's not in E2
- May name regular expressions
- Ex:
  - `letter ::= a | b | ... | z`
  - `digit  ::= 0 | 1 | ... | 9`
  - `alphanum ::= letter | num`
- *Recursive definitions prohibited*
- Produce simple regular expression from named regular expressions by "macro expansion"

Examples

- Identifiers
  - `ident ::= letter ( digit | letter)*`
- Integer constants
  - `integer ::= digit+`
  - `sign ::= + | -`
  - `signed_int ::= [sign] integer`
- Real numbers
  - `real ::= signed_int [fraction] [exponent]`
  - `fraction ::= . digit+`
  - `exponent ::= (E | e) signed_int`

More Examples

- String and character constants
  - `string ::= " char* "`
  - `character ::= ' char '`
  - `char ::= not(" | ' | \) | escape`
  - `escape ::= \(" | ' | \ | n | r | t | v | b | a )`
- White space
  - `whitespace ::= <space> | <tab> | <newline> | comment`
  - `comment ::= /* not(*/) */`

Meta-Rules

- Consider a program defined as:
  - `program ::= (token | whitespace)*`
  - `token ::= ident | integer | real | …`
- Then consider how to tokenize 'hi2bob'
  - <ident: 'hi2bob'> ?
  - <ident: 'hi', integer: '2', ident: 'bob'>
  - Or six separate tokens?
- All are legal according to the grammar, but the choice does matter – the ambiguity isn't desirable here
- Usually apply an extra rule such as "longest sequence wins"

Initial MiniJava lexical specification

```
Program ::= (Token | Whitespace)*
Token ::= ID | Integer | ReservedWord | Operator |
           Delimiter
ID ::= Letter (Letter | Digit)*
Letter ::= a | ... | z | A | ... | Z
Digit ::= 0 | ... | 9
Integer ::= Digit+
ReservedWord::= class | public | static | extends |
        void | int | boolean | if | else |
       while|return|true|false| this | new | String
       | main | System.out.println
Operator ::= + | - | * | / | < | <= | >= | > | == |
       != | && | !
Delimiter ::= ; | . | , | = | ( | ) | { | } | [ | ]
```


Building Scanners with REs

- Convert regular expressions into finite state automata (FSA)
- Convert FSA into a scanner implementation
  - By hand into a collection of procedures
  - Mechanically using a table-driven scanner

Finite State Automata

- On whiteboard; see book

(Non)Determinism

- FSA can be deterministic (DFA) or nondeterministic (NFA)
- Deterministic: always know which edge to take
  - At most one arc leaving a state with a given symbol
  - No ε arcs
- Nondeterministic: may need to guess or explore multiple paths, choosing the right one later
- Regular expressions map naturally to NFAs
- Hard to produce scanner code from NFAs but easy to produce from DFAs

A Solution

- Cool algorithm to translate any NFA to a DFA
  - Proves that NFAs aren't any more expressive
    - But… what might happen?
  - Can be done by hand or automatically

1. Convert RE to NFA
2. Convert NFA to DFA
3. Convert DFA to code
   – [can then minimize DFA]

Trivia: Who invented this cool algorithm?

RE => NFA: construct inductively

- On whiteboard; see book

NFA => DFA

- Problem: NFA can "choose" among alternative paths, while DFA must pick only one path
- Solution: subset construction
  - Each state in the DFA represents the set of states the NFA could possibly be in

Subset Construction

- On whiteboard; see book

Tokens

- Every “final” symbol of a DFA emits a token
- Tokens are the internal compiler names for the lexemes

`==`	becomes **`equal`**
`(`	becomes **`leftParen`**
`private`	becomes **`private`**

- Also, there may be additional data representing the attribute

DFA => Code

- Option 1: Implement by hand using procedures
  - one procedure for each token
  - each procedure reads one character
  - choices implemented using if and switch statements
- Pros: straightforward to write, fast
- Cons
  - a fair amount of tedious work
  - may have subtle differences from the language specification

DFA => code

- Option 2: use tool to generate table driven parser
  - Rows: states of DFA
  - Columns: input characters
  - Entries: Go to next state, Accept token, Error
- Pros
  - Convenient
  - Exactly matches specification, if tool generated
- Cons
  - “Magic”
  - Table lookups may be slower than direct code

Automatic Scanner Generation

- We use the jflex tool to automatically create a scanner from a specification file, Scanner/minijava.jflex
  - We use the CUP tool to automatically create a parser from a specification file, Parser/minijava.cup, which also generates all of the code for the token classes used in the scanner, via the Symbol class
- The MiniJava Makefile automatically rebuilds the scanner (or parser) whenever its specification file changes

Symbol Class

- Lexemes are represented as instances of **`class Symbol`**

```
class Symbol {
Int sym;      // which token class?
Object value; // attribute info for this lexeme
...
}
```

- A constant is defined for each token class in the **`sym`** helper class

```
class sym {
    static int CLASS = 1;
    static int IDENTIFIER = 2;
    static int COMMA = 3;
 ...
}
```

- Can use this in printing code for **`Symbol`**s; see **`symbolToString`** in **`minijava.jflex`**

Token Declarations

- Declare new token classes in **`Parser/minijava.cup`**, using **`terminal`** declarations
  - If **`Symbol`** stores attribute data, then its type must be defined
- Examples

```
/* reserved words: */
terminal CLASS, PUBLIC, STATIC, EXTENDS; …
/* operators: */
terminal PLUS, MINUS, STAR, SLASH, EXCLAIM; …
/* delimiters: */
terminal OPEN_PAREN, CLOSE_PAREN; …
terminal EQUALS, SEMICOLON, COMMA, PERIOD; …
/* tokens with values: */
terminal String IDENTIFIER;
terminal Integer INT_LITERAL; …
```


jflex Token Specifications

- Helper definitions for character classes and regular expressions (e.g., **`letter = [a-z A-Z]`**, **`eol = [\r\n]`**)
- Token patterns are defined as **`regexp { Java stmt }`**
- **`regexp`** may include
  - a string literal in double-quotes, e.g. "class", "<="
  - a reference to a named helper, in braces, e.g., **`{letter}`**
  - a character list or range, in square brackets ,e.g., **`[a-z A-Z]`**
  - a negated character list or range, e.g., **`[^\r\n]`**
  - **`.`** (which matches any single character)
  - concatenation, alternation, Kleene * and +, optional, grouping

jflex Token: accept action

- for a simple token
  `return symbol(sym.CLASS);`
- for a token with attribute data
  `return symbol(sym.CLASS,yytext()); stringyytext()`
- empty for whitespace

Some lies I told

- Sometimes the parser calls the scanner and requests a token, rather than creating a token stream and passing it to the parser
- Sometimes there is some language information useful in the scanner; for example, the parser may wish the scanner to distinguish between names that are types and names that are variables (in C++ and Java, for example)
  – But the scanner doesn't know how things are declared; so this can complicate the dependences