


CSE 582 – Compilers

Languages, Automata, Regular Expressions & Scanners
Hal Perkins
Autumn 2002


10/3/2002 © 2002 Hal Perkins & UW CSE B-1



Agenda for Today

- n Basic concepts of formal grammars (review)
- n Regular expressions
- n Lexical specification of programming languages
- n Using finite automata to recognize regular expressions
- n Scanners and Tokens


10/3/2002 © 2002 Hal Perkins & UW CSE B-2



Announcements

- n New on the web since last time
 - n Class discussion list
 - n Links to Java resources, including several development environments
 - n Video of Tuesday's lecture
 - n Sorry, slide transitions didn't make it – will fix
 - n Homework 1 – warmup problems on regular expressions
 - n Due Tuesday 6 pm electronically, or bring to class if you're attending at UW


10/3/2002 © 2002 Hal Perkins & UW CSE B-3



Programming Language Specs

- n Since the 1960s, the syntax of every significant programming language has been specified by a formal grammar
 - n First done in 1959 with BNF (Backus-Naur Form or Backus-Normal Form) used to specify the syntax of ALGOL 60
 - n Borrowed from the linguistics community (Chomsky?)


10/3/2002 © 2002 Hal Perkins & UW CSE B-4



Grammar for a Tiny Language

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

10/3/2002 © 2002 Hal Perkins & UW CSE B-5



Productions

- n The rules of a grammar are called *productions*
- n Rules contain
 - n Nonterminal symbols: grammar variables (*program*, *statement*, *id*, etc.)
 - n Terminal symbols: concrete syntax that appears in programs (a, b, c, 0, 1, if, (, ...)
- n Meaning of
 - $nonterminal ::= \langle sequence\ of\ terminals\ and\ nonterminals \rangle$
 - In a derivation, an instance of *nonterminal* can be replaced by the sequence of terminals and nonterminals on the right of the production
- n Often, there are two or more productions for a single nonterminal – can use either at different times

10/3/2002 © 2002 Hal Perkins & UW CSE B-6

Alternative Notations

- There are several syntax notations for productions in common use; all mean the same thing
 - $ifStmt ::= if (expr) stmt$
 - $ifStmt \rightarrow if (expr) stmt$
 - $\langle ifStmt \rangle ::= if (\langle expr \rangle) \langle stmt \rangle$

10/3/2002 © 2002 Hal Perkins & UW CSE B-7

Example Derivation

```

program ::= statement | program statement
statement ::= assignStmt | ifStmt
assignStmt ::= id = expr ;
ifStmt ::= if ( expr ) stmt
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
  
```

a = 1 ; if (a + 1) b = 2 ;

10/3/2002 © 2002 Hal Perkins & UW CSE B-8

Parsing

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

10/3/2002 © 2002 Hal Perkins & UW CSE B-9

Parsing & Scanning

- n In real compilers the recognizer is split into two phases
 - n Scanner: translate input characters to tokens
 - n Also, report lexical errors like illegal characters and illegal symbols
 - n Parser: read token stream and reconstruct the derivation

```

graph LR
  source --> Scanner[Scanner]
  Scanner -- tokens --> Parser[Parser]
            
```

10/3/2002 © 2002 Hal Perkins & UW CSE B-10

Characters vs Tokens (review)

- n Input text


```
// this statement does very little
if (x >= y) y = 42;
```
- n Token Stream


IF	LPAREN	ID(x)	GEQ	ID(y)
RPAREN	ID(y)	BECOMES	INT(42)	SCOLON

10/3/2002 © 2002 Hal Perkins & UW CSE B-11

Why Separate the Scanner and Parser?

- n Simplicity & Separation of Concerns
 - n Scanner hides details from parser (comments, whitespace, input files, etc.)
 - n Parser is easier to build; has simpler input stream
- n Efficiency
 - n Scanner can use simpler, faster design
 - n (But still often consumes a surprising amount of the compiler's total execution time)


10/3/2002 © 2002 Hal Perkins & UW CSE B-12



Tokens

- n Idea: we want a distinct token kind (lexical class) for each distinct terminal symbol in the programming language
 - n Examine the grammar to find these
- n Some tokens may have attributes
 - n Examples: integer constant token will have the actual integer (17, 42, ?) as an attribute; identifiers will have a string with the actual id


10/3/2002 © 2002 Hal Perkins & UW CSE B-13



Typical Tokens in Programming Languages

- n Operators & Punctuation
 - n + - * / () { } [] ; : : < <= == != ! ...
 - n Each of these is a distinct lexical class
- n Keywords
 - n if while for goto return switch void ...
 - n Each of these is also a distinct lexical class (not a string)
- n Identifiers
 - n A single ID lexical class, but parameterized by actual id
- n Integer constants
 - n A single INT lexical class, but parameterized by int value
- n Other constants, etc.

10/3/2002 © 2002 Hal Perkins & UW CSE B-14



Principle of Longest Match

- n In most languages, the scanner should pick the longest possible string to make up the next token if there is a choice
- n Example



```
return foobar != hohum;
```

 should be recognized as 5 tokens

RETURN	ID(foobar)	NEQ	ID(hohum)	SCOLON
--------	------------	-----	-----------	--------

 not more (i.e., not parts of words or identifiers, or ! and = as separate tokens)


10/3/2002 © 2002 Hal Perkins & UW CSE B-15



Review: Languages & Automata Theory (in one slide)

- Alphabet: a finite set of symbols
- String: a finite, possibly empty sequence of symbols from an alphabet
- Language: a set, often infinite, of strings
- Finite specifications of (possibly infinite) languages
 - Automaton – a recognizer; a machine that accepts all strings in a language (and rejects all other strings)
 - Grammar – a generator; a system for producing all strings in the language (and no other strings)
- A language may be specified by many different grammars and automata
- A grammar or automaton specifies only one language


10/3/2002 © 2002 Hal Perkins & UW CSE B-16



Regular Expressions and FAs

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

10/3/2002 © 2002 Hal Perkins & UW CSE B-17



Regular Expressions

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

10/3/2002 © 2002 Hal Perkins & UW CSE B-18

Fundamental REs

re	$L(re)$	Notes
a	$\{ a \}$	Singleton set, for each a in Σ
ϵ	$\{ \epsilon \}$	Empty string
\emptyset	$\{ \}$	Empty language

10/3/2002 © 2002 Hal Perkins & UW CSE B-19

Operations on REs

re	$L(re)$	Notes
rs	$L(r)L(s)$	Concatenation
$r s$	$L(r) \cup L(s)$	Combination (union)
r^*	$L(r)^*$	0 or more occurrences (Kleene closure)

- Precedence: * (highest), concatenation, | (lowest)
- Parentheses can be used to group REs as needed

10/3/2002 © 2002 Hal Perkins & UW CSE B-20

Abbreviations

- The basic operations generate all possible regular expressions, but there are common abbreviations used for convenience. Typical examples:

Abbr.	Meaning	Notes
r^+	(rr^*)	1 or more occurrences
$r?$	$(r \epsilon)$	0 or 1 occurrence
$[a-z]$	$(a b \dots z)$	1 character in given range
$[abxyz]$	$(a b x y z)$	1 of the given characters

10/3/2002 © 2002 Hal Perkins & UW CSE B-21

Examples

<i>re</i>	Meaning
+	single + character
!	single ! character
=	single = character
!=	2 character sequence
<=	2 character sequence
hogwash	7 character sequence

10/3/2002 © 2002 Hal Perkins & UW CSE B-22

More Examples


<i>re</i>	Meaning
[abc]+	
[abc]*	
[0-9]+	
[1-9][0-9]*	
[a-zA-Z][a-zA-Z0-9_]*	

10/3/2002 © 2002 Hal Perkins & UW CSE B-23

Abbreviations

- Many systems allow abbreviations to make writing and reading definitions easier
 - name ::= *re*
- Restriction: abbreviations may not be circular (recursive) either directly or indirectly

10/3/2002 © 2002 Hal Perkins & UW CSE B-24




Example

- Possible syntax for numeric constants


```

digit ::= [0-9]
digits ::= digit+
number ::= digits ( . digits )?
           ( [eE] (+ | -)? digits )?
      
```


10/3/2002 © 2002 Hal Perkins & UW CSE B-25



Recognizing REs

- Finite automata can be used to recognize strings generated by regular expressions
- Can build by hand or automatically
 - Not totally straightforward, but can be done systematically
 - Tools like Lex, Flex, and JLex do this automatically, given a set of REs

10/3/2002 © 2002 Hal Perkins & UW CSE B-26



Finite State Automaton

- A finite set of states
 - One marked as initial state
 - One or more marked as final states
 - States sometimes labeled or numbered
- A set of transitions from state to state
 - Each labeled with symbol from Σ , or ϵ
- Operate by reading input symbols (usually characters)
 - Transition can be taken if labeled with current symbol
 - ϵ -transition can be taken at any time
- Accept when final state reached & no more input
 - Scanner slightly different – accept longest match even if more input
- Reject if no transition possible or no more input and not in final state (DFA)

10/3/2002 © 2002 Hal Perkins & UW CSE B-27

Example: FSA for "cat"

10/3/2002 © 2002 Hal Perkins & UW CSE B-28

DFA vs NFA

- Deterministic Finite Automata (DFA)
 - No choice of which transition to take under any condition
- Non-deterministic Finite Automata (NFA)
 - Choice of transition in at least one case
 - Accept if some way to reach final state on given input
 - Reject if no possible way to final state

10/3/2002 © 2002 Hal Perkins & UW CSE B-29

FAs in Scanners

- Want DFA for speed (no backtracking)
- Conversion from regular expressions to NFA is easy
- There is a well-defined procedure for converting a NFA to an equivalent DFA

10/3/2002 © 2002 Hal Perkins & UW CSE B-30

From RE to NFA: base cases

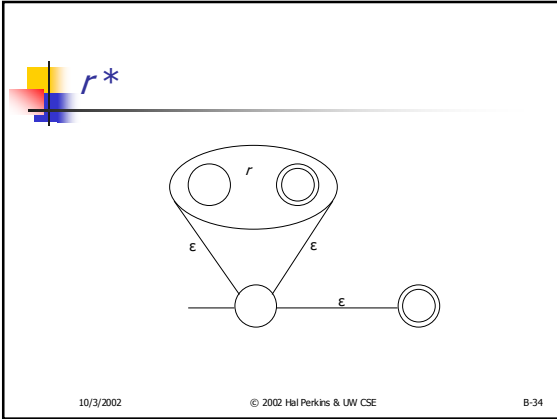
10/3/2002 © 2002 Hal Perkins & UW CSE B-31

rs

10/3/2002 © 2002 Hal Perkins & UW CSE B-32

r | s

10/3/2002 © 2002 Hal Perkins & UW CSE B-33



From NFA to DFA

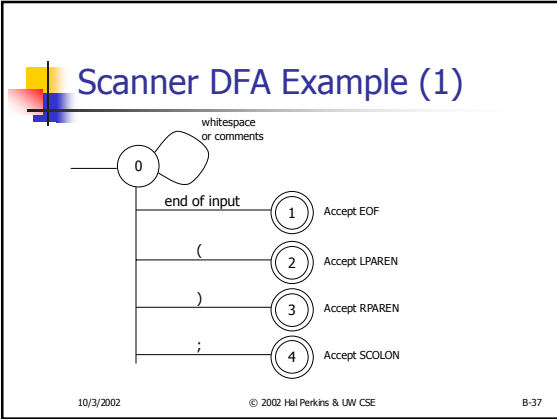
- n Subset construction
 - n Construct a DFA from the NFA, where each DFA state represents a *set* of NFA states
- n Key idea
 - n The state of the DFA after reading some input is the set of *all* states the NFA could have reached after reading the same input
- n Algorithm: example of a fixed-point computation
- n If NFA has n states, DFA has at most 2^n states
 - n => DFA is finite, can construct in finite # steps
- n Resulting DFA may have more states than needed
 - n See the book for construction and minimization details

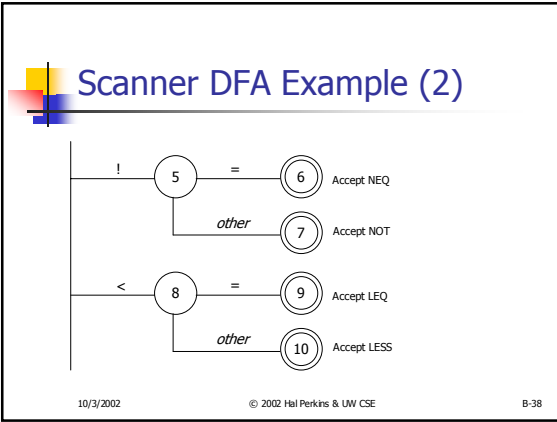
10/3/2002 © 2002 Hal Perkins & UW CSE B-35

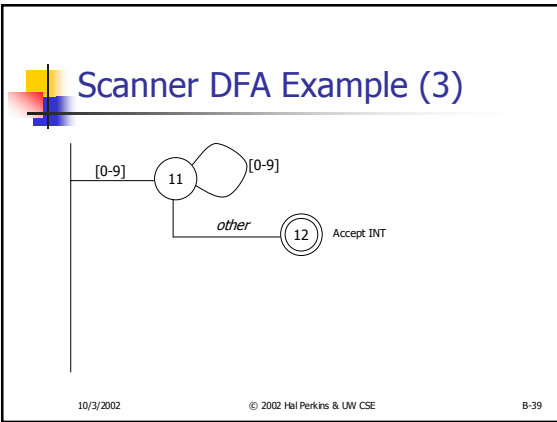
Example: DFA for hand-written scanner

- n Idea: show a hand-written DFA for some typical programming language constructs
 - n Then use to construct hand-written scanner
- n Setting: Scanner is called whenever the parser needs a new token
 - n Scanner stores current position in input
 - n Starting there, use a DFA to recognize the longest possible input sequence that makes up a token and return that token

10/3/2002 © 2002 Hal Perkins & UW CSE B-36







Scanner DFA Example (4)

```

graph LR
    13((13)) -- "[a-zA-Z]" --> 13
    13 -- "other" --> 14(((14)))
    style 14 stroke-width:4px
  
```

- Strategies for handling identifiers vs keywords
 - Hand-written scanner: look up identifier-like things in table of keywords to classify (good application of perfect hashing)
 - Machine-generated scanner: generate DFA with appropriate transitions to recognize keywords
 - Lots of states, but efficient (no extra lookup step)

10/3/2002 © 2002 Hal Perkins & UW CSE B-40

Implementing a Scanner by Hand – Token Representation

- A token is a simple, tagged structure


```

public class Token {
    public int kind;           // token's lexical class
    public int intVal;        // integer value if class = INT
    public String id;         // actual identifier if class = ID
    // lexical classes
    public static final int EOF = 0; // "end of file" token
    public static final int ID = 1;  // identifier, not keyword
    public static final int INT = 2; // integer
    public static final int LPAREN = 4;
    public static final int SCOLN = 5;
    public static final int WHILE = 6;
    // etc. etc. etc. ...
}

```

10/3/2002 © 2002 Hal Perkins & UW CSE B-41

Simple Scanner Example

```

// global state and methods
static char nextch; // next unprocessed input character

// advance to next input char
void getch() { ... }

// skip whitespace and comments
void skipWhitespace() { ... }

```

10/3/2002 © 2002 Hal Perkins & UW CSE B-42



Scanner getToken() method

```
// return next input token
public Token getToken() {
    Token result;

    skipWhiteSpace();

    if (no more input) {
        result = new Token(Token.EOF); return result;
    }

    switch(nextch) {
        case '(': result = new Token(Token.LPAREN); getch(); return result;
        case ')': result = new Token(Token.RPAREN); getch(); return result;
        case ';': result = new Token(Token.SCOLON); getch(); return result;

        // etc. ...
    }
}
```

10/3/2002

© 2002 Hal Perkins & UW CSE

B-43



getToken() (2)

```
case '!': // ! or !=
    getch();
    if (nextch == '=') {
        result = new Token(Token.NEQ); getch(); return result;
    } else {
        result = new Token(Token.NOT); return result;
    }

case '<': // < or <=
    getch();
    if (nextch == '=') {
        result = new Token(Token.LEQ); getch(); return result;
    } else {
        result = new Token(Token.LESS); return result;
    }

// etc. ...
```

10/3/2002

© 2002 Hal Perkins & UW CSE

B-44



getToken() (3)


```
case '0': case '1': case '2': case '3': case '4':
case '5': case '6': case '7': case '8': case '9':
    // integer constant
    String num = nextch;
    getch();
    while (nextch is a digit) {
        num = num + nextch; getch();
    }
    result = new Token(Token.INT, Integer(num).intValue());
    return result;

...
}
```

10/3/2002

© 2002 Hal Perkins & UW CSE

B-45




getToken (4)

```

case 'a': ... case 'z':
case 'A': ... case 'Z': // id or keyword
string s = nextch; getch();
while (nextch is a letter, digit, or underscore) {
    s = s + nextch; getch();
}
if (s is a keyword) {
    result = new Token(keywordTable.getKind(s);
} else {
    result = new Token(Token.ID, s);
}
return result;

```


10/3/2002 © 2002 Hal Perkins & UW CSE B-46



Project Notes

- n For the CSE582 project, use a lexical analyzer generator like Lex (i.e., JLex for Java or something equivalent)
- n JLex is designed to work with CUP, a parser generator
 - n JLex needs to return an instance of the class Symbol, the token class CUP expects as input
 - n CUP knows about the lexical classes (of course) and will generate a class named sym with those constants. These need to be used by the scanner.

10/3/2002 © 2002 Hal Perkins & UW CSE B-47



Coming Attractions

- n Homework this weekend: paper exercises on regular expressions, etc.
- n Next week: first part of the compiler assignment – the scanner
- n Next topic: parsing
 - n Will do LR parsing first, since we need it for the project
 - n Good time to start reading ch. 3.

10/3/2002 © 2002 Hal Perkins & UW CSE B-48
