Review for the Midterm
COMS W4115
Prof. Stephen A. Edwards
Fall 2007
Columbia University
Department of Computer Science

The Midterm
70 minutes
4–5 problems
Closed book
One sheet of notes of your own devising
Comprehensive: Anything discussed in class is fair game
Little, if any, programming.
Details of ANTLR/C/Java/Prolog/ML syntax not required
Broad knowledge of languages discussed

Topics
Structure of a Compiler
Scripting Languages
Scanning and Parsing
Regular Expressions
Context-Free Grammars
Top-down Parsing
Bottom-up Parsing
ASTs
Name, Scope, and Bindings

Compiling a Simple Program
int gcd(int a, int b)
{
    while (a != b) {
        if (a > b) a -= b;
        else b -= a;
    }
    return a;
}

What the Compiler Sees
int gcd(int a, int b)
{
    while (a != b) {
        if (a > b) a -= b;
        else b -= a;
    }
    return a;
}

Lexical Analysis Gives Tokens
int gcd(int a, int b)
{
    while (a != b) {
        if (a > b) a -= b;
        else b -= a;
    }
    return a;
}

A stream of tokens. Whitespace, comments removed.

Parsing Gives an AST

Semantic Analysis Resolves Symbols

Translation into 3-Address Code

Text file is a sequence of characters

Idealized assembly language w/ infinite registers
Generation of 80386 Assembly

gcd: pushl %ebp
movl %esp, %ebp
movl 8(%ebp), %eax
% Load a from stack
movl 12(%ebp), %edx
% Load b from stack
.L8: cmpl %edx, %eax
% while (a != b)
je .L3 % if (a < b)
jle .L5
% a -= b
subl %edx, %eax
jmp .L8
.L5: subl %eax, %edx
% b -= a
jmp .L8
.L3: leave
% Restore SP, BP
ret

Scanning and Automata

Describing Tokens

Alphabet: A finite set of symbols
Examples: {0, 1}, {A, B, C, ...}, ASCII, Unicode

String: A finite sequence of symbols from an alphabet
Examples: ε (the empty string), Stephen, αβγ

Language: A set of strings over an alphabet
Examples: ∅ (the empty language), {1, 11, 111, 1111}, all English words, strings that start with a letter followed by any sequence of letters and digits

Operations on Languages

Let \( L = \{ \epsilon, wo \}, M = \{ \text{man}, \text{men} \} \)

Concatenation: Strings from one followed by the other
\( LM = \{ \text{man}, \text{men}, \epsilon \text{man}, \epsilon \text{men}, \epsilon \text{manman}, \epsilon \text{manmen}, \epsilon \text{manmenman}, \epsilon \text{manmenmen}, ... \} \)

Union: All strings from each language
\( L \cup M = \{ \epsilon, \text{wo}, \text{man}, \text{men} \} \)

Kleene Closure: Zero or more concatenations
\( M^* = \{ \epsilon, M, MM, MMM, ... \} = \{ \epsilon, \text{man}, \text{men}, \text{manman}, \text{manmen}, ... \} \)

The Language induced by an NFA

An NFA accepts an input string \( x \) if there is a path from the start state to an accepting state that "spells out" \( x \).

Regular Expressions over an Alphabet \( \Sigma \)

A standard way to express languages for tokens.
1. \( \epsilon \) is a regular expression that denotes \{\( \epsilon \}\}
2. If \( a \in \Sigma \), \( a \) is an RE that denotes \{a\}
3. If \( r \) and \( s \) denote languages \( L(r) \) and \( L(s) \),
   - \( (r) (s) \) denotes \( L(r) \cup L(s) \)
   - \( (r) (s) \) denotes \( \{ tu : t \in L(r), u \in L(s) \} \)
   - \( (r)^* \) denotes \( \cup_{i=0}^{\infty} L^i (L^0 = \emptyset \text{ and } L^i = LL^{i-1}) \)

Translating REs into NFAs

Example: translate \( (a|b)^* abb \) into an NFA

Translating REs into NFAs

Example: translate \( a[b]^* \) into an NFA

Show that the string "010101" is accepted.

Show that the string "\( aabb \)" is accepted.
Simulating NFAs

Problem: you must follow the “right” arcs to show that a string is accepted. How do you know which arc is right?
Solution: follow them all and sort it out later.
“Two-stack” NFA simulation algorithm:
1. Initial states: the $\epsilon$-closure of the start state
2. For each character $c$,
   - New states: follow all transitions labeled $c$
   - Form the $\epsilon$-closure of the current states
3. Accept if any final state is accepting
Simulating an NFA: \( aabb \)

Deterministic Finite Automata

Restricted form of NFAs:
- No state has a transition on \( \epsilon \)
- For each state \( s \) and symbol \( a \), there is at most one edge labeled \( a \) leaving \( s \).

Differs subtly from the definition used in COMS W3261 (Sipser, *Introduction to the Theory of Computation*).

Very easy to check acceptance: simulate by maintaining current state. Accept if you end up on an accepting state. Reject if you end on a non-accepting state or if there is no transition from the current state for the next symbol.

Deterministic Finite Automata

ELSE: "else" ; ELSEIF: "elseif" ; IF: "if" ; ID: 'a'..'z' ('a'..'z' | '0'..'9') * ; NUM: ('0'..'9')+ ;

Building a DFA from an NFA

Subset construction algorithm

Simulate the NFA for all possible inputs and track the states that appear.

Each unique state during simulation becomes a state in the DFA.

Subset construction for (\( a|b \))^*abb (1)

Subset construction for (\( a|b \))^*abb (2)

Subset construction for (\( a|b \))^*abb (3)
**Subset construction for \((a|b)^*abb\)**

**Grammars and Parsing**

**Ambiguous Grammars**

A grammar can easily be ambiguous. Consider parsing

\[ 3 - 4 \times 2 + 5 \]

with the grammar

\[ e \rightarrow e + e | e - e | e \times e | e / e \]

**Fixing Ambiguous Grammars**

Original ANTLR grammar specification

```antlr
expr : expr '+' expr
    | expr '-' expr
    | expr '*' expr
    | expr '/' expr
    | NUMBER ;
```

Ambiguous: no precedence or associativity.

**Assigning Precedence Levels**

Split into multiple rules, one per level

```antlr
expr : expr '+' term
    | expr '-' term
    | term ;
```

**Assigning Associativity**

Make one side or the other the next level of precedence

```antlr
term : term '*' atom
    | term '/' atom
    | atom ;
```

**A Top-Down Parser**

```c
stmt : 'if' expr 'then' expr
    | 'while' expr 'do' expr
    | expr ':'= expr ;

expr : NUMBER | '(' expr ')' ;
```

**Writing LL(k) Grammars**

Cannot have left-recursion

```c
expr : expr '+' term | term ;
```

becomes

```c
AST expr() {
  switch (next-token) {
    case NUMBER : expr(); /* Infinite Recursion */
    ...
  }
}
```

**Writing LL(1) Grammars**

Cannot have common prefixes

```c
expr : ID '(' expr ')'
    | ID '=' expr ;
```

becomes

```c
AST expr() {
  switch (next-token) {
    case ID : match(ID); match('('); expr(); match(')');
    case ID : match(ID); match('='); expr();
  }
}
```
Eliminating Common Prefixes

Consolidate common prefixes:

\[
\text{expr} : \text{expr} \ ' + ' \text{term} \\
| \text{expr} \ ' - ' \text{term} \\
| \text{term} \\
\]

becomes

\[
\text{expr} : (\text{term} + \text{term}) \\
| \text{term} \\
\]

Rightmost Derivation

1: \( e \rightarrow t + e \)  
2: \( e \rightarrow t \)  
3: \( t \rightarrow t + t \)  
4: \( t \rightarrow t \)  

A rightmost derivation for \( t + t + t \): 

Basic idea of bottom-up parsing: construct this rightmost derivation backward. 

The outlined terms are what we are expanding, not handles.

Shift-reduce Parsing

1: \( e \rightarrow t + e \) stack input action  
2: \( e \rightarrow t \)  
3: \( t \rightarrow t + t \)  
4: \( t \rightarrow t \)  

Scan input left-to-right, looking for handles. 
An oracle tells what to do.

Eliminating Left Recursion

Understand the recursion and add tail rules

\[
\text{expr} : \text{expr} \ ' + ' \text{term} | \ ' - ' \text{term} \\
| \text{term} \\
\]

becomes

\[
\text{expr} : \text{term} \text{expr} \\
\text{expr} : \ ' + ' \text{term} \text{expr} | \ ' - ' \text{term} \text{expr} \\
| \text{term} \\
\]

LR Parsing

1: \( e \rightarrow t + e \) stack input action  
2: \( e \rightarrow t \)  
3: \( t \rightarrow t + t \)  
4: \( t \rightarrow t \)  

1. Look at state on top of stack and the next input token 
2. To find the next action 
3. In this case, shift the token onto the stack and go to state 1.

Handle Hunting

Parsing ⇒ reducing handles in a right-sentential form 

The trick: we can recognize handles with a finite automaton—the parse table.

Bottom-up Parsing

Consolidate common prefixes:

\[
\text{expr} : \text{expr} \ ' + ' \text{term} \\
| \text{expr} \ ' - ' \text{term} \\
| \text{term} \\
\]

becomes

\[
\text{expr} : \text{term} \text{expr} \\
\text{expr} : \ ' + ' \text{term} \text{expr} | \ ' - ' \text{term} \text{expr} \\
| \text{term} \\
\]

LR Parsing

1: \( e \rightarrow t + e \) stack input action  
2: \( e \rightarrow t \)  
3: \( t \rightarrow t + t \)  
4: \( t \rightarrow t \)  

Action is reduce with rule 4 
\( (t \rightarrow t) \). The right side is removed from the stack to reveal state 3. The goto table in state 3 tells us to go to state 5 when we reduce a \( : \) 


text=\{random:乮seg1乯\}
LR Parsing

1: \( e \rightarrow t + e \)
2: \( e \rightarrow t \)
3: \( t \rightarrow Id \cdot t \)
4: \( t \rightarrow Id \)

action goto

Id * Id + Id $ shift, goto 1
Id + Id $ shift, goto 3
Id $ shift, goto 4
$ reduce w/ 4
$ reduce w/ 2
$ reduce w/ 1
$ accept

Constructing the SLR Parse Table

The states are places we could be in a reverse-rightmost derivation. Let's represent such a place with a dot.

1: \( e \rightarrow t + e \)
2: \( e \rightarrow t \)
3: \( t \rightarrow Id \cdot t \)
4: \( t \rightarrow Id \)

Say we were at the beginning (\( e \)). This corresponds to

\( e' \rightarrow e \) The first is a placeholder. The second are the two possibilities when we're just before \( e \).

\( e \rightarrow t + e \) \( e \rightarrow t \) when we're just before \( e \). The last two are the two possibilities when we're just before \( t \).

Names, Objects, and Bindings

Activation Records

Nested Subroutines in Pascal

Symbol Tables in Tiger