Review for the Midterm

COMS W4115
Prof. Stephen A. Edwards
Spring 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
Control-flow

Compiling a Simple Program

```c
int gcd(int a, int b)
{
    while (a != b) {
        if (a > b) a -= b;
        else b -= a;
    }
    return a;
}
```

What the Compiler Sees

```c
int g c d ( i n t a , i n t b ) {
    while ( a != b ) {
        if ( a > b ) a -= b ;
        else b -= a ;
    }
    return a ;
}
```

Lexical Analysis Gives Tokens

```c
int gcd(int a, int b) {
    while (a != b) {
        if (a > b) a -= b;
        else b -= a;
    }
    return a;
}
```

Text file is a sequence of characters

Parsing Gives an AST

```c
int gcd(int a, int b)
{
    while (a != b) {
        if (a > b) a -= b;
        else b -= a;
    }
    return a;
}
```

Abstract syntax tree built from parsing rules.

Semantic Analysis Resolves Symbols

```c
int gcd(int a, int b)
{
    while (a != b) {
        if (a > b) a -= b;
        else b -= a;
    }
    return a;
}
```

Types checked; references to symbols resolved

Translation into 3-Address Code

```c
L0: sne $1, a, b
    seq $0, $1, 0
    % while (a != b)
    sl $3, b, a
    seq $2, $3, 0
    % if (a < b)
    btrue $2, L4 % if (a < b)
    sub a, a, b % a = b
    jmp L5
L4: sub b, b, a % b = a
L5: jmp L0
L1: ret a
```

Idealized assembly language w/ infinite registers
### Generation of 80386 Assembly

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

---

### Scanning and Automata

#### Operations on Languages

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

**Concatenation**: Strings from one followed by the other

$L \cdot M = \{ \text{man, men, woman, women} \}$

**Union**: All strings from each language

$L \cup M = \{ \epsilon, \text{wo, man, men} \}$

**Kleene Closure**: Zero or more concatenations

$M^* = \{ \epsilon, M, MM, MMM, \ldots \} = \{ \epsilon, \text{man, men, manman, manmen, menman, menmen, manmanman, manmanmen, manmenman, } \ldots \}$

---

#### 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 $\bigcup_{i=0}^{\infty} L^i$ ($L^0 = \emptyset$ and $L^1 = LL^{-1}$)

---

#### Describing Tokens

**Alphabet**: A finite set of symbols

Examples: $\{ 0, 1 \}, \{ A, B, C, \ldots, Z \}$, ASCII, Unicode

**String**: A finite sequence of symbols from an alphabet

Examples: $\emptyset$ (the empty string), Stephen, $\alpha\beta\gamma$

**Language**: A set of strings over an alphabet

Examples: $\emptyset$ (the empty language), $\{ 1, 11, 111, 1111 \}$, all English words, strings that start with a letter followed by any sequence of letters and digits

---

#### Nondeterministic Finite Automata

**"All strings containing an even number of 0's and 1's"**

**1. Set of states $S$:** $\{ A, B, C, D \}$
2. **Set of input symbols $\Sigma$:** $\{ 0, 1 \}$
3. **Transition function $\sigma$:** $S \times \Sigma \rightarrow 2^S$
4. **Start state $s_0$**: $A$
5. **Set of accepting states $F$:** $\{ A \}$

---

#### Translating REs into NFAs

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

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: $aabb$, Start

Simulating an NFA: $aabb$, $\epsilon$-closure

Simulating an NFA: $aa\cdot bb$

Simulating an NFA: $aa\cdot bb$, $\epsilon$-closure

Simulating an NFA: $aab\cdot b$

Simulating an NFA: $aab\cdot b$, $\epsilon$-closure
Simulating an NFA: \textit{aabb}.

Deterministic Finite Automata

ELSE: "else" ;
ELSEIF: "elseif" ;

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

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

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

Deterministic Finite Automata

Restrict 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 \(s\).

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.
Grammars and Parsing

Fixing Ambiguous Grammars

Original ANTLR grammar specification

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

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

```antlr
atom
  : NUMBER ;
```

Still ambiguous: associativity not defined

Assigning Associativity

Make one side or the other the next level of precedence

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

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

```antlr
atom
  : NUMBER ;
```

A Top-Down Parser

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

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

```java
AST stmt() {
  switch (next-token) {
    case "if" : match("if"); expr(); match("then"); expr();
    case "while" : match("while"); expr(); match("do"); expr();
    case NUMBER or "" : expr(); match("="); expr();
  }
}
```

Writing LL(k) Grammars

Cannot have left-recursion

```antlr
expr
  : ID '(' expr ')' 
    | ID '=' expr
```

```antlr
AST expr() {
  switch (next-token) {
    case ID : match(ID); match('('); expr(); match(')');
    case ID : match(ID); match('='); expr();
  }
}
```

Writing LL(1) Grammars

Cannot have common prefixes

```antlr
expr
  : ID '(' expr ')' 
```

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

Consolidate common prefixes:

\[
\text{expr} \\
: \text{expr} \ 't' \ \text{term} \\
| \text{expr} \ 't' \ \text{term} \\
| \text{term} \\
| \\
\]

becomes

\[
\text{expr} \\
: \text{expr} \ ('t' \ \text{term} | 't' \ \text{term}) \\
| \text{term} \\
| \\
\]

Eliminating Left Recursion

Understand the recursion and add tail rules

\[
\text{expr} \\
: \text{expr} \ ('t' \ \text{term} | 't' \ \text{term}) \\
| \text{term} \\
| \\
\]

becomes

\[
\text{expr} \\
: \text{term} \ \text{expr} \\
| \text{expr} \ 't' \ \text{term} \ \text{expr} \\
| 't' \ \text{term} \ \text{expr} \\
| */ \text{nothing} /* \\
| \\
\]

Bottom-up Parsing

Rightmost Derivation

Here, I've drawn a box around each symbol to expand.

A rightmost derivation for \( \text{id} + \text{id} + \text{id} \):

\[
1: \text{id} \rightarrow \text{id} + \text{id}
2: \text{id} \rightarrow \text{id}
3: \text{id} \rightarrow \text{id}
4: \text{id} \rightarrow \text{id}
\]

This is a reverse rightmost derivation for \( \text{id} + \text{id} + \text{id} \).

Each highlighted section is a handle.

Taken in order, the handles build the tree from the leaves to the root.

Handles

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

Shift-reduce Parsing

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

LR Parsing

1. Look at state on top of stack
2. and the next input token
3. to find the next action
4. In this case, shift the token onto the stack and go to state 1.
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:  $t \rightarrow id + t$
4:  $t \rightarrow id$

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

- $e' \rightarrow e$: The first is a placeholder.
- $e \rightarrow t + e$: The second are the two possibilities when we're just before $e$.
- $t \rightarrow id + t$: The last two are the two possibilities when we're just before $t$.

Names, Objects, and Bindings

Activation Records

- argument 1
- argument 2
- return address
- local variables
- temporaries/arguments
- frame pointer
- stack pointer
- growth of stack

Activation Records

- Return Address
- Frame Pointer
- A's variables
- B's variables
- C's variables

Symbol Tables in a Functional Lang.

let

- var $n := 8$
- var $x := 3$
- function $sqr(a:\text{int}) = a \ast a$
- type $ia = \text{array of int}$
- in
- $n := sqr(x)$
- end

Control-Flow

Nested Subroutines in Pascal

procedure A;
var $a : \text{integer}$;
procedure B;
var $b : \text{integer}$;
procedure C;
var $c : \text{integer}$;
begin .. end
procedure D;
var $d : \text{integer}$;
begin .. end
procedure E;
var $e : \text{integer}$;
begin .. end

Record for A

- static link $a$

Record for B

- static link $b$

Record for C

- static link $c$

Record for D

- static link $d$

Record for E

- static link $e$

Record for A

- static link $a$

Record for B

- static link $b$

Record for C

- static link $c$

Record for D

- static link $d$

Record for E

- static link $e$

Symbol Tables in a Functional Lang.

int A()
{
    int $x$;
    B();
}

int B()
{
    int $y$;
    C();
}

int C()
{
    int $z$;
}
**Side-effects**

```c
int x = 0;
int foo() { x += 5; return x; }
int a = foo() + x + foo();
GCC sets a=25.
Sun's C compiler gave a=20.
C says expression evaluation order is implementation-dependent.
```

**Misbehaving Floating-Point Numbers**

1e20 + 1e-20 = 1e20
1e-20 ≪ 1e20
(1 + 9e-7) + 9e-7 ≠ 1 + (9e-7 + 9e-7)
9e-7 ≪ 1, so it is discarded, however, 1.8e-6 is large enough
1.00001(1.000001 − 1) ≠ 1.00001 · 1.000001 − 1.00001 · 1
1.00001 · 1.000001 = 1.0000100001 requires too much intermediate precision.

**Multi-way Branching**

```c
switch (s) {
  case 1: one(); break;
  case 2: two(); break;
  case 3: three(); break;
  case 4: four(); break;
}
Switch sends control to one of the case labels. Break terminates the statement.
```

**Implementing multi-way branches**

```c
switch (s) {
  case 1: one(); break;
  case 2: two(); break;
  case 3: three(); break;
  case 4: four(); break;
}
Obvious way:
if (s == 1) { one(); }
else if (s == 2) { two(); }
else if (s == 3) { three(); }
else if (s == 4) { four(); }
Reasonable, but we can sometimes do better.
```

**Applicative- and Normal-Order Evaluation**

```c
int p(int i) { printf("\n", i); return i; }
void q(int a, int b, int c) {
  int total = a;
  printf("\n", b);
  total += c;
}
q( p(1), 2, p(3) );
Applicative: arguments evaluated before function is called.
Result: 1 3 2
Normal: arguments evaluated when used.
Result: 1 2 3
```

**Nondeterminism**

Nondeterminism is not the same as random:
Compiler usually chooses an order when generating code.
Optimization, exact expressions, or run-time values may affect behavior.
Bottom line: don’t know what code will do, but often know set of possibilities.
```c
int p(int i) { printf("\n", i); return i; }
int q(int a, int b, int c) {
  int total = a;
  printf("\n", b);
  total += c;
}
q( p(1), p(2), p(3) );
Will not print 5 & 7. It will print one of
1 2 3 1 3 2 2 1 3 2 1
```

**Implementing Inheritance**

```c
C++
class Shape {
  double x, y;
};
class Box : Shape {
  double h, w;
};
Equivalent C
```
Virtual Functions

class Shape {
  virtual void draw(); // Invoked by object's class
}; // not its compile-time type.
class Line : public Shape {
  void draw();
};
class Arc : public Shape {
  void draw();
};
Shape *s[10];
s[0] = new Line;
s[1] = new Arc;
s[0]->draw(); // Invoke Line:draw()
s[1]->draw(); // Invoke Arc:draw()

Virtual Functions

The Trick: Add a "virtual table" pointer to each object.
struct A {
  int x;
  virtual void Foo();
  virtual void Bar();
};
struct B : A {
  int y;
  virtual void Foo();
  virtual void Baz();
};
A a1, a2; B b1;

Virtual Functions

Exceptions

A high-level replacement for C's setjmp/longjmp.
struct Except {
  void baz() { throw Except; } // Reraise
  void bar() { baz(); } // Reraise
  void foo() {
    try {
      bar();
    } catch (Except e) {
      printf("oops");
    }
  }
}

One Way to Implement Exceptions

try {
  push(Ex, Handler);
  throw Ex;
  pop();
  goto Exit;
} catch (Except e) {
  Handler:
  foo();
  Exit:
}
push() adds a handler to a stack
pop() removes a handler
throw() finds first matching handler
Problem: imposes overhead even with no exceptions

Implementing Exceptions Cleverly

Real question is the nearest handler for a given PC.

```
1 void foo() {
  try {
    bar();
  } catch (Ex1 e) {
    H1:
  } catch (Ex2 e) {
    H2:
  }

3–5 H1
6–9 H2
13–14 Reraise
```