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

Lexical Analysis Gives Tokens

Text file is a sequence of characters

Parsing Gives an AST

Abstract syntax tree built from parsing rules.

Semantic Analysis Resolves Symbols

Types checked; references to symbols resolved.

Translation into 3-Address Code

Idealized assembly language w/ infinite registers
**Generation of 80386 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
% 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, ... Z\}, ASCII, Unicode

**String**: A finite sequence of symbols from an alphabet
Examples: \(\epsilon\) (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

**Operations on Languages**

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

**Concatenation**: Strings from one followed by the other
\(LM = \{man, men, woman, women\}\)

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

**Kleene Closure**: Zero or more concatenations
\(M^* = \{\epsilon, M, MM, MMM, \ldots\} = \{\epsilon, man, men, manman, manmen, menman, menmen, manmanman, manmenmen, mannemanman, \ldots\}\)

**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 \(L(r) \cup L(s)\)
   - \(rs\) denotes \(\{tu : t \in L(r), u \in L(s)\}\)
   - \(r^*\) denotes \(\bigcup_{i=0}^{\infty} L^i\) (\(L^0 = \emptyset\) and \(L^i = LL^{i-1}\))

**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: $aa-bb$

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

Simulating an NFA: $aab-b$

Simulating an NFA: $aab-b$, $\epsilon$-closure
Simulating an NFA: $aabb$.

Deterministic Finite Automata

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

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

Deterministic Finite Automata

IF: "if" ;
ID: 'a'..'z' ('a'..'z' | '0'..'9') ;
NUM: ('0'..'9')+ ;

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

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 $(a|b)^*abb$ (3)
Ambiguous Grammars

A grammar can easily be ambiguous. Consider parsing

3 - 4 * 2 + 5

with the grammar

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

Fixing Ambiguous Grammars

Original ANTLR grammar specification

\[
\text{expr} : \text{expr} '+' \text{expr} | \text{expr} '-' \text{expr} | \text{expr} '*' \text{expr} \\
| \text{expr} '/' \text{expr} | \text{NUMBER};
\]

Ambiguous: no precedence or associativity.

Assigning Precedence Levels

Split into multiple rules, one per level

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

\[
\text{term} : \text{term} ' * ' \text{atom} | \text{term} '/' \text{atom} | \text{atom} ;
\]

\[
\text{atom} : \text{NUMBER} ;
\]

Still ambiguous: associativity not defined.

Assigning Associativity

Make one side or the other the next level of precedence

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

\[
\text{term} : \text{term} ' * ' \text{atom} | \text{term} '/' \text{atom} | \text{atom} ;
\]

\[
\text{atom} : \text{NUMBER} ;
\]

A Top-Down Parser

\[
\text{stmt} : 'if' \text{expr} 'then' \text{expr} | 'while' \text{expr} 'do' \text{expr} | \text{expr} ;
\]

\[
\text{expr} : \text{NUMBER} | '(' \text{expr} ')' ;
\]

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

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

becomes

\[
\text{expr} (\text{expr}(); /* Infinite Recursion */
\]

Writing LL(1) Grammars

Cannot have common prefixes

\[
\text{expr} : \text{ID} '(' \text{expr} ')' | \text{ID} '=' \text{expr}
\]

becomes

\[
\text{expr} (\text{match(ID) match(\) ; expr() match(=) ; expr();
\]

}\)

\[
\text{match(ID) match(\) ; expr() match(=) ; expr();
\]

}\)

\[
\text{match(ID) match(\) ; expr() match(=) ; expr();
\]

}\)
Eliminating Common Prefixes

Consolidate common prefixes:

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

becomes

```plaintext
expr
  : expr ('+' term | '-' term )
  | term
  ;
```

Eliminating Left Recursion

Understand the recursion and add tail rules

```plaintext
expr
  : expr ('+' term | '-' term )
  | term
  ;
```

becomes

```plaintext
expr : term exprt ;
exprt : '+' term exprt
| '-' term exprt
| '/' term exprt
| '*' term exprt
```

Bottom-up Parsing

**Rightmost Derivation**

1. \( e \rightarrow t + e \)
2. \( e \rightarrow t \)
3. \( t \rightarrow \text{Id} + t \)
4. \( t \rightarrow \text{Id} \)

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

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

```plaintext
\( t + \text{Id} \)
\( \text{Id} + \text{Id} \)
\( \text{Id} + \text{Id} + \text{Id} \)
```

**Shift-reduce Parsing**

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

```
1 : e \rightarrow t + e
2 : e \rightarrow t
3 : t \rightarrow \text{Id} + t
4 : t \rightarrow \text{Id}
```

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

Each highlighted section is a handle.

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

**LR Parsing**

```
1 : e \rightarrow t + e
2 : e \rightarrow t
3 : t \rightarrow \text{Id} + t
4 : t \rightarrow \text{Id}
```

```
0 : s1
1 : r4 r4 r3 r4
2 : r2 r2 r2 r2
3 : s1
4 : s1
5 : r3 r3 r3 r3
6 : r1 r1 r1 r1
7 : acc
```

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.

Action is reduce with rule 4

(\( t \rightarrow \text{Id} \)). 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 \( t \):

- stack
- input
- action

```
0 : s1
1 : r4 r4 r3 r4
2 : r2 r2 r2 r2
3 : s1
4 : s1
5 : r3 r3 r3 r3
6 : r1 r1 r1 r1
7 : acc
```

- stack
- input
- action

```
0 : s1
1 : r4 r4 r3 r4
2 : r2 r2 r2 r2
3 : s1
4 : s1
5 : r3 r3 r3 r3
6 : r1 r1 r1 r1
7 : acc
```

- stack
- input
- action

```
0 : s1
1 : r4 r4 r3 r4
2 : r2 r2 r2 r2
3 : s1
4 : s1
5 : r3 r3 r3 r3
6 : r1 r1 r1 r1
7 : acc
```
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 (\( \cdot \)). This corresponds to 
\( e' \rightarrow \cdot \) The first is a placeholder. The second are the two possibilities when we’re just before \( e \). The last two are the two possibilities when we’re just before \( t \).

### Constructing the SLR Parsing Table

\begin{align*}
S0: & e' \rightarrow \cdot e \\
S1: & t \rightarrow Id + t \\
S2: & t \rightarrow Id \\
S3: & t \rightarrow Id + t \\
S4: & t \rightarrow Id \\
S5: & t \rightarrow Id \\
S6: & e \rightarrow t + e \\
S7: & e' \rightarrow \cdot e
\end{align*}

### Names, Objects, and Bindings

- **Object1** binding **Name1**
- **Object2** binding **Name2**
- **Object3** binding **Name3**
- **Object4** binding **Name4**

### Activation Records

- argument 2
- argument 1
- return address
- old frame pointer
- local variables
- temporaries/arguments

- frame pointer
- stack pointer
- \( \downarrow \) growth of stack

### Activation Records

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

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

```
int C() {
  int z;
}
```

### Nested Subroutines in Pascal

```pascal
procedure A;
  procedure B;
    procedure C;
    begin .. end
  begin .. end
  procedure D;
  begin C end
  begin D end
  procedure E;
  begin B end
  begin E end
```

### Symbol Tables in Tiger

```
let
var n := 8
var x := 3
function sqr(a:int)
  = a * a
in
  type ia = array of int
  n := sqr(x)
end
```