Compiler Construction
CS 541

<table>
<thead>
<tr>
<th>Mr</th>
<th>Raphael</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dr</td>
<td>Rafi</td>
</tr>
<tr>
<td>Prof</td>
<td>Finkel</td>
</tr>
<tr>
<td></td>
<td>Goldstein</td>
</tr>
</tbody>
</table>

www.cs.uky.edu/~raphael/courses/CS541.html

MultiLab:

ssh pen.cs.uky.edu
cor.
faw.
Compiler outputs

- pure machine code
- augmented machine code
  - system calls
- virtual machine code
  - interpreted machine
  - JVM
  - portability
  - code size (register assignment)

Output representations

- assembler, for later processing
- relocatable binary (machine instructions)
  - modular compilation
- absolute binary
Structure of a compiler

1. Scanner (lexer)
2. Tokens
3. Parser
4. AST (abstract syntax tree)
5. Code generator

Program → Scanner → Tokens → Parser → AST → Code generator → Assembler code

Code:
```
if (a < 39) 3
```

- if
- statement
- cond: a < 39
- then: 3
- else
- expr: a
- int: 39
- symbol: a
- integer literal: 39
- identifier: a
Special considerations

1) Programming languages

- Pass by name (Algol 60)
- Dynamic-sized arrays
- Nested name scopes

2) Anonymous functions

- First-class functions
- Iterators (Python generators)
- Automatic object reclamation

Computer architecture

- How many registers?
- Cache principle: frequent operations should be fast.
- What operations are expensive?
- Virtual method dispatch exceptions
effects of caches + paging

3) Specialty compilers

debugging support

IDE integrated development environment

optimizing compilers

retargetable

---

Chapter 2: Adding calculator

types: `integer`, `float`

keywords: `+`, `-`, `*`, `=`, `if`, `else`

variables: lowercase Roman single

letters

Syntax: Context-free grammar

Backus-Naur Form (BNF)
Ambiguity in grammars leads to multiple parse trees:

```
if (a)
  if (b)
    S_1
  else
    S_2
```

Scanner:
translate a stream of characters
into a stream of tokens.

```
Stream:
  nextItem (stream): next item
  peek (stream): no modify
  advance (stream): next item modifies
  match (stream, item): expects the given item, advances stream, error if not
```
token: has a type (like id, sinum) ⑦
has a semantic value (like 'a', 39)

Choices: reserved words: each one a
different type ② call them all id, and distinguish in the
semantic value. ③ call them all type reserved

Hard-wired scanner (for ac)
Alternative: generated scanner

Regular expressions.
Adequate for defining token syntax.
Input to automatic scanner generators.

1) any string (including empty)
2) concatenation of 2 reg. exps.
3) alternation of 2 reg. exps.

Example: a*b* alternation
          a | b* AS
          AS ← concatenation
Formal-Language Hierarchy

<table>
<thead>
<tr>
<th>Use</th>
<th>Language Type</th>
<th>Formalism</th>
<th>Automaton</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tokens(Scanner)</td>
<td>Regular</td>
<td>Regular expressions</td>
<td>Finite-State Automaton</td>
</tr>
<tr>
<td>Syntax(Parser)</td>
<td>Context-Free</td>
<td>BNF</td>
<td>Push-Down Automaton</td>
</tr>
<tr>
<td></td>
<td>Context-Sensitive</td>
<td>CS Grammars</td>
<td>Linear-Bounded</td>
</tr>
<tr>
<td></td>
<td>Typed</td>
<td>Post Productions</td>
<td>Turing Machine</td>
</tr>
</tbody>
</table>

Parser

\[
\text{stream of tokens} \xrightarrow{\text{parser}} \text{abstract syntax tree}
\]

Many approaches
- Recursive descent
  - L(1)
  - L(3)
  - \ldots

\[
f(b, c) \cdot a = 5 \cdot b = c + 32 \quad p \times b \quad f
\]

Program Needs:
- \text{Dcl}
- \text{Dcls}
  - \text{Dcl}
    - \text{Dcls}
      - id
      - \text{Dcl}
    - id
  - \text{stmts}
Limitations of scanner + parser
enforce type rules.
disambiguate the meanings of some
constructs:

\[ x \cdot y \cdot z \quad x \cdot y \cdot z \quad \text{package class field class field field} \]

resolve overloaded operators
\[ a + b \quad (\ldots) + (\ldots) \]
int, float, String
Abstract Syntax Tree (AST)

Program

Decls list of Decl

Decl

Stmts

Assign

Print class:

Add

LHS

num 2

RHS

id b

class Decl

Decl type (for i)

name (like b)

Class Hierarchy

Stmt

Assign

LHS: id b

RHS: Expr

Print

id: 10

Decls list

Program

stmts:Stmts

decls: Decl

Expr

type (for i)

id

name

num value

Fnum value

Operation

op1: Expr

op2: Expr

operation: (+ or -)
Semantic Analysis
1) Symbol table to record types and name scopes, (ST)
   For ac, trivial: array indexed by 'a'...'z'
   element has type: unknown, int, float
2) Recursive walk through AST invoking check() on each node in tree
   Modifying the ST during declarations
   Updating the AST as needed for type conversion
   Updating the type field in expressions.
   Checking that types are consistent.
   Checking other semantic constraints: reachability
   exception consistency.

Code generation
1) Recursive walk through AST invoking codeGen() on each node

Scanner: table-driven (as opposed to hard-coded)

```
program -> stream of chars
          -> Scanner
          -> stream of tokens
```
Token specification: regular expressions

\( \emptyset \): no valid strings
\( \lambda \): empty string
\( \Sigma \): letters in alphabet
\( \Sigma^* \): a string containing that letter
\( \Sigma^+ \): concatenation of 2 REs (no symbol)
\( \Sigma^\ast \): resulting strings: cross product; one from each
\( \Sigma^\ast \): concatenation of 2 REs (symbol \( \ast \))
\( \Sigma \cup \): union of two sets
\( \Sigma^+ \): closure operation.

Examples:

Alphabet \( \Sigma = \{ a, b, c \} \)
\( \emptyset \): set of strings
\( \lambda \): empty
\( \Sigma^+ \): \{ "", "\}
Useful facts

- The set of strings specified by a RE is a regular set.
- Every finite set of strings is a regular set.
- Every regular set has multiple REs that describe it.

**Notation**

- If $A$ is a set of characters, $\mathsf{not}(A)$ refers to the other characters.
- If $S$ is a set of strings, $\mathsf{not}(S)$ is also regular.

\[ S^k = S \cdot S \cdot S \cdot \ldots \cdot S \]
String: element of the language specified by a RE.

Examples

Java comment: //
RE: // (not \n)* \n\n---

Decimal literal
D = 0123 ... 19
D+ . D+ \equiv DD+. DD*

Integer literal, optionally signed

(-)? D+ 4
(- | +)? D+ 34
(-1 | +2) D+ (-1+12) D+

Comment with delimiters ##

## ((#(>) not(#)))* ##

Example: ## hi ## there ##

Fortran-like real literals: must have digits in at least one side.

( D+ . D* ) | ( D* . D+ ) 4.7
(p+. ) | (.D+) | (D+.D+)

(D+. D*) | (.D+)

Identifier, with chars, but not adjacent, not frontal, not find ...
First char may not be a digit.

L Letter; D Digit

L (L|D)* ( _ (L|D)* )*

Mickler: L ((-|+)(L|D) )*  

Hashing: Θ(1) searching and insertion
(actually Θ(log n): have to look at entire key)
Java provides an interface Map<K,V>

Map<String, String> myHash =
    new HashMap<String, String>();

List<Integer> myList =
    new LinkedList<Integer>();
Hashing: given a key, apply a function $h(k)$ returning an integer. Use that integer to index an array. Store lookup that (key, value) pair at that location in the array.

Collisions: multiple keys can have same index.

Example: birthday paradox.

Resolution:
- open addressing: find another place
- external chaining: array points to a secondary structure, typically a list.

Finite-state automata (FSA)

Simple computer
- finite set of states -- circles
  - one is start state
  - one or more states are final (accepting)
- transitions between states labeled by letters in $\Sigma$
The diagram shows a state transition graph with the following states:

- **Start**: State 1
- **Final**: State 4

The transitions are:
- From state 1 to state 2 on input a
- From state 2 on input b
- From state 2 on input c
- From state 3 on input c
- From state 4 on input a

There is a loop from state 3 back to state 2 on input a.

The annotations include:
- Error: Anything
- **Not(a)**

The expressions annotated are:
- (abc)^+ c^* ✗
- (abc^+)^+ ✓