Compiler Construction
CS 541

Mr
Mr
Prof

Raphael
Rafi

Finkel
Goldstein

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

Program → #2 scanner → #3 parser

lexer

Symbol table

#1

Fast

Symbol table

#5

code generator

AST

#4 tree

semantic checker

assembler code

code:

if (a < 39) { 3

symbol identifier

integer literal

expr

id a

39 int

then

cond

else

if statement
Special considerations

1) Programming by
   pass by name (Algol 60)
   dynamic-sized arrays
   nested name scopes
   anonymous functions
   first-class functions
   iterators (Python generators)
   automatic object reclamation

2) Computer architecture
   how many registers?
   cache principle: frequent
   operations should be fast.
   What operations are expensive?
   virtual method dispatch,
   exceptions
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

\[
\text{if (a)} \\
\quad \text{if (b)} \\
\quad \text{else} \\
\]

\[
S_1, \quad S_2, \quad S_2, \quad S_2, \quad S_2
\]

Scanner:

translate a stream of characters into a stream of tokens.

\[
\text{Stream: item \text{\_peek (stream): next item}} \\
\quad \text{item \text{\_advance (stream): next item}} \\
\quad \text{match (stream, item): error if not}
\]
token: has a type (like `id`, `int`) has a semantic value (like `a`, `39`)

Choices: reserved words: each one a different type (call them all `id`, and distinguish in the semantic value) (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. exprs.
3) alternation of 2 reg. exprs.

Example: `$a^* b^*$` alternation
$A = \gamma \mid Ka \mid AS$ concatenation
Formal-Language Hierarchy

**Use**
- Tokens (Scanner)
- Syntax (Parser)

**Language Type**
- Regular
- Context-free
- Context-sensitive
- Typed

**Formalism**
- Regular expressions
- BNF
- CS grammar
- Post productions

**Automaton**
- Finite-state automaton
- Push-down automaton
- Linear-bounded
- Turing machine

**Parser**
- stream of tokens → parser
- abstract syntax tree

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

**Program Needs**
- Decl Decls
  - \( \text{f} \), \( \text{id} \)
  - \( \text{Decl} \), \( \text{Decl} \)
  - \( \text{InstDecl} \), \( \text{id} \)

**Example**
- \( f b i a c = 5 b = c + 32 \)
Limitations of scanner + parser

- enforce type rules.
- disambiguate the meanings of some constructs:
  - `x, y, Z`  vs  `x, y, z`  vs  `x, y, Z`
    - `package class field`  vs  `class field field`

- resolve overloaded operators:
  - `a + b`  vs  `(....) + (....)`
    - `int, float, String`
Abstract Syntax Tree (AST)

Program

Decls
- list of Decl

Class Decl
- host decl b
- output decl a

Starts

Assign
- LHS
- RHS
- id
- operation
- from
- to

Print:
- class
- stmt
- id
- a
- from
- 3.2

Class Hierarchy

Decl
- type (for i)
- name (like b)

Start

Assign
- LHS: id
- RHS: Expr

Print
- id: 10

Stmts:

Decls:

Expr
- type (for i)
  - id
  - name
  - Lnum
  - value
  - from
  - value
  - Operation
    - op1: Expr
    - op2: Expr
    - operation: (+ or -)

Program
- stmts: Starts
- decls: Decl
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 hardware) hard-coded

\[
\text{program} \xrightarrow{\text{stream of chars}} \text{Scanner} \xrightarrow{\text{stream of tokens}} \text{Table}
\]
Token specification: regular expressions

$\emptyset$: no valid strings

$\lambda$: empty string

$\Sigma$: letters in alphabet

$\varepsilon$: a string containing that letter.

Concatenation of 2 REGs (no symbol)

resulting strings: cross product, one from each alternate of 2 REGs (symbol $|$)

resulting strings: union of two sets.

closure operation:

$\varepsilon$: $\emptyset$ or more concatenations

$+$: 1 or more concatenation

parentheses for grouping

escape convention for metachars.

Examples:

Alphabet $\Sigma = \{a, b, c\}$

$\emptyset$: set of strings

$\lambda$: empty

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

$S^k = SSSSS\ldots S$
String: element of the language specified by a RE.

Examples

Java comment: //

RE: // (not \n)* \n meta chars

Decimal literal

\( D = 0 1 2 \ldots 9 \)

\( D^+ \cdot D^* = DD* \cdot DD^* \)

Integer literal, optionally signed

\( (-)? \, D^* \quad ^{\text{eg}} 34 \)

\( (-\mid+)? \, D^+ \)

\( (-l\mid+\lambda) \, D^+ \)

Comment with delimiters

```
#\# ( (#(?) not(#)) )* #\#  
```

Example:

```
# hi # there #
```

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

\( (D^+ \cdot D^*) \mid (D^* \cdot D^+) \quad 4.7 \)
identifier, with _ chars, but not adjacent _, not # _ #, # # #, first char may not be a digit.

L Letter; D digit

\[ L (L | D)^* (\_ (L | D)^*)^* \]

Mickler: \[ L ((-12)(L | D))^* \]

Hashing: \( O(1) \) searching and insertion

(actually \( O(\log n) \); have to look at entire key)

Java provides an interface `Map<K, V>`

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

```java
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 $(k, v)$ 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 $\Sigma$. 
\[ X_i = x_0 y_0 \]

\[ Y_i = x^i y_0 = x_0 y_0 \wedge y_0. \]

\[ X_2 = x^2 y_1 = x_0 y_0 \wedge x_0 \]

grading assignments

look for lines: // grader:

file: grade

returned file is a ZIP file

Nice Java features

\[ \text{for } (\text{var} : \text{collection}) \]

\[ \text{3} \]

\[ \text{switch can take string conditions.} \]
switch (String expr) {
    case "foo": break;
    default: 
}

Short story: Tokens are described by REs.
1) REs are encoded into non-deterministic finite-state automata (NFA).
2) NFA are converted to DFA (deterministic).
3) The DFA are described by tables
   state x input -> action x state
4) A simple program can run the DFA.

Shorter story:
1) Write a set of REs describing tokens
2) Let a scanner generator (e.g., flex) build the tables
3) Let a pre-built scanner run the tables.

Complexities (if strange tokens)
1) Escape " within a string literal
2) Eagerness
   Pascal 3..4
   Ada 'ab'
   FORTRAN DO 200 I = 1,2...
built by an automatic process: lex, flex, jflex
alternative: automatically build a hard-wired scanner. Likely to be faster.
Transducer recognizes strings in a regular language and outputs some semantic value.

Natural-language processing:

- tables $\rightarrow$ lexeme (noun) table
  - plural
- $\rightarrow$ lexeme (verb) table
  - 3rd sg present

Each transition is associated with an action. (Build a growing string, guess at meaning, construct value of a literal)

---

Scanner generator (lex, flex, jflex)

Input file: defines tokens and actions * .jlex
Output: a table (embedded in code) and associated code: yylex() jflex: yylex()

Compile and link that output with a client (P2jasm)

(driver, parser)

Input file (jflex)

Subroutines
- 9.2%
Declarations
- 9.2%
REs and actions

Shared with client:
- Token codes (small integers)
Although:

\[
\text{list} \langle \ldots \rangle \var = \text{LinkedList} \langle \ldots \rangle
\]

Sometimes:

\[
\text{LinkedList} \langle \ldots \rangle \var
\]

// need push/\langle\rangle when you need methods in implementation type not available in interface type

\[
(\text{LinkedList} \var).\text{push}(\ldots)
\]

Lex

declarations:

short hand for RES:

\[
\text{DIGIT} = 0123\ldots9
\]

\[
[0-9]
\]

\[
[013-9]
\]

\[
\text{VAR} = [a-ehj-0q-2]
\]

\[
[aa\,b]
\]

\[
[^ab]
\]

\[
\Sigma - [ab]
\]

\[
\text{ALLCHARS} = [^a]^b
\]

\cdot
Foo = "ab\_c"

abc \rightarrow \text{equiv}

"abc" \equiv [abc]

\text{case is significant}

\text{\texttt{ignorecase} \rightarrow makes all lower = upper.}

[a] = [aA]

[bB][cC][eE][gG][iI][tT][uU][nN]

\text{meta characters}

( ) \ast \star \text{^} \text{\^} \text{\^} \text{\^} \text{\^} \text{\^} \text{\^} \text{\^}

\text{Rules}

D \rightarrow \{ \text{return (INTLIT)}; \}

3 \text{\text{processing code: we invoke yytext() to a String value containing what was matched by RE.}}
D+ 1  D+ 3  ε
3
    return (FLOAT_LIST);
}

"{" 3  return (OPENBRACE);

jflex creates: Yylex.java
javac -classpath .;classes:*.jar *.java
provided by jflex

Advice:

Last rule
ε 3  return (ERROR);
E DF taken is automatic. (taken value = 0)

Subroutine section
Introduce declarations of vars and functions, and classes.

"A diagnostic": a compiler-generated error message.

Handle Pascal's \texttt{3..4}

\texttt{D+ | D+/"."} \varepsilon \texttt{int lit}
\texttt{even if looking ahead has 2 dots}.
If there is an error in input (not less) entered an error state.
backup until the scanner is in a final state.
if all the way back, discard 1 character, diagnostic, continue.
Example (6x - 90) : N ""

If not Lex CLA, readc : not table-driven scanner.

Identifies

1) if not block-structured, scanner enter ids into ST.
2) if block-structured, copy the string in "string space" and return a pointer.

String space

<table>
<thead>
<tr>
<th>foo bar</th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
</tr>
<tr>
<td>^</td>
</tr>
</tbody>
</table>
simpler: duplicate string values.

3) scanner to enter in to ST that is block-structured controlled by parser.
4) scanner returns strings to parser, which uses them.
Literals

numbers: convert to internal representation.
C: atoi() atof()
Java: Integer.parseInt(s)
range? use higher precision and range checks.
String: expand escaped characters.
\t \r \n \v 

Ambiguity
\((x, y)\)  
\text{typedef}
\int (x, y)
what to call x? \{identifier \} \{typeDeclaration\}

Reserved words
special REs dealing with them
\Rightarrow \text{large FSA.}
Scanner can have a table of reserved words

Compiler directives
C: \#include
\Rightarrow \{stack of input files\}
\{recursively invoke scanner\}
conditional compilation + macros
C: \#if \{\} \{\} \{separate pass\}
\text{\#define}: \{\} \{\} \{pass 0\}

Unicode escapes:
Java: \u05b3 \{pass 0\}
Listing: \text{\#had to intersperse diagnostics.}
End of file (EOF)

special token, returned multiple times

Multi-character lookahead, error recovery

FORTRAN: DO 10 I = 1, 10

Pascal: 3..4

↑

general method

buffer characters.

error state => back up to first

accepting state

if none, discard first character

Unbounded string

"""

""

=> cascading errors

instead: special-purpose error rule

Speed

(Rules for optimization)

1) if fast enough, leave it.
2) Computers will get faster, wait a year.
3) Check for best algorithms/data structures
4) use a profiling tool to localize

where program is spending time.

Optimize right there.

invoke compiler with optimization on.

)

1) Use a scanner generator.
2) Hard-coded: block read operations.
   double buffering: keep 2 buffers available.
Converting REs to PFA:

\[ \text{REs} \rightarrow \text{NDFA} \rightarrow \text{DFA} \rightarrow \text{REs} \]

\[ \emptyset \]

\[ \text{a} \]

\[ \text{A} \text{B} \]

\[ (D^+ \cdot D^*) | (D^+ \cdot D^*) = (D^* \cdot D^*) | (D^* \cdot D^*) \]
Subset construction.

NFA

DFA

\[ b^*a(a/b(a/b|^\lambda)|\lambda) \]

\[ b^*a^* b^*a(a/b)|b^*a^*a \]

\[ babb \]

\[ a(alb)^* \]

\[ aabb \]

\[ \begin{array}{c|c|c|c}
  a & b \\
  \hline
  X & Y \\
  1 & 2 & 3 & 4
\end{array} \]
Build a RE from a DFA.

Builds a single RE.

Actual scanner must deal with many REs.

Shown building DFA for a single RE.

Build distinguishing final states by their active final state can retain an appropriate name.