CSE 401: Introduction to Compiler Construction

Goals:

- learn principles & practice of language implementation
  - brings together theory & pragmatics of previous courses
  - understand compile-time vs. run-time processing
- study interactions among:
  - language features
  - implementation efficiency
  - compiler complexity
  - architectural features
- gain more experience with object-oriented design & Java
- gain more experience working on a team

Prerequisites: 322, 326, 341, 378

Text: *Engineering a Compiler*

Sign up on course mailing list!

Course Outline

Compiler front-ends:

- lexical analysis (scanning): characters → tokens
- syntactic analysis (parsing): tokens → abstract syntax trees
- semantic analysis (typechecking): annotate ASTs

Midterm

Compiler back-ends:

- intermediate code generation: ASTs → intermediate code
- target code generation: intermediate code → target code
  - run-time storage layout
  - target instruction selection
  - register allocation
- optimizations

Final

Project

Start with compiler for MiniJava, written in Java

Add:

- comments
- floating-point values
- arrays
- static (class) variables
- for loops
- break statements
- and more

Completed in stages over the quarter

Strongly encourage working in a 2-person team on project

- but only if joint work, not divided work

Grading based on:

- correctness
- clarity of design & implementation
- quality of test cases

Grading

Project: 40% total
Homework: 20% total
Midterm: 15%
Final: 25%

Homework & projects due at the **start of class**

3 free late days, per person, for the whole quarter

- thereafter, 25% off per calendar day late

An example compilation

Sample (extended) MiniJava program: `Factorial.java`

```
// Computes 10! and prints it out
class Factorial {
    public static void main(String[] a) {
        System.out.println(
            new Fac().ComputeFac(10));
    }
}

class Fac {
    // the recursive helper function
    public int ComputeFac(int num) {
        int numAux = 0;
        if (num < 1)
            numAux = 1;
        else
            numAux = num * this.ComputeFac(num-1);
        return numAux;
    }
}
```


First step: lexical analysis

“Scanning”, “tokenizing”

Read in characters, clump into **tokens**

- strip out whitespace & comments in the process

Specifying tokens: regular expressions

Example:

```
Ident    ::= Letter AlphaNum*
Integer  ::= Digit+
AlphaNum ::= Letter | Digit
Letter   ::= 'a' | ... | 'z' | 'A' | ... | 'Z'
Digit    ::= '0' | ... | '9'
```


Second step: syntactic analysis

“Parsing”

Read in tokens, turn into a tree based on syntactic structure

- report any errors in syntax

Specifying syntax: context-free grammars

EBNF is a popular notation for CFG's

Example:

```
Stmt    ::= if ( Expr ) Stmt [else Stmt]
          | while ( Expr ) Stmt
          | ID = Expr;
          | ...
Expr    ::= Expr + Expr | Expr < Expr | ...
          | ! Expr
          | Expr . ID ( [Expr {, Expr}] )
          | ID
          | Integer | ...
          | (Expr)
          | ...
```

EBNF specifies *concrete syntax* of language

Parser usually constructs tree representing *abstract syntax* of language

Third step: semantic analysis

"Name resolution and typechecking"

Given AST:

- figure out what declaration each name refers to
- perform typechecking and other static consistency checks

Key data structure: symbol table

- maps names to info about name derived from declaration
- tree of symbol tables corresponding to nesting of scopes

Semantic analysis steps:

1. Process each scope, top down
2. Process declarations in each scope into symbol table for scope
3. Process body of each scope in context of symbol table

Fourth step: intermediate code generation

Given annotated AST & symbol tables, translate into lower-level intermediate code

Intermediate code is a separate language

- Source-language independent
- Target-machine independent

Intermediate code is simple and regular
$\Rightarrow$ good representation for doing optimizations

Might be a reasonable target language itself, e.g. Java bytecode

Example

```
int Fac.ComputeFac(*? this, int num) {
   int T1, numAux, T8, T3, T7, T2, T6, T0;
   numAux := 1;
   T0 := 1;
   T1 := num < T0;
   ifnonzero T1 goto L0;
   T2 := 1;
   T3 := num - T2;
   T6 := Fac.ComputeFac(this, T3);
   T7 := num * T6;
   numAux := T7;
   goto L2;
   label L0;
   T8 := 1;
   numAux := T8;
   label L2;
   return numAux;
}
```

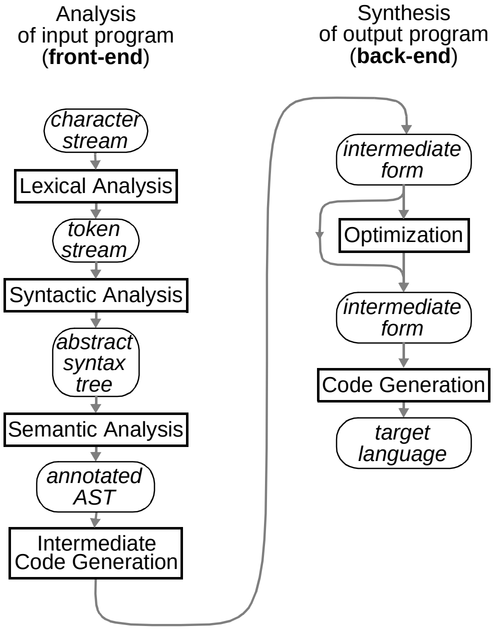
Fifth step: target (machine) code generation

Translate intermediate code into target code

Need to do:

- instruction selection: choose target instructions for (subsequences of) intermediate code instructions
- register allocation: allocate intermediate code variables to machine registers, spilling excess to stack
- compute layout of each procedure's stack frame & other run-time data structures
- emit target code

Summary of compiler phases



Ideal: many front-ends, many back-ends sharing one intermediate language

Other language processing tools

Compilers translate the input language into a different, usually lower-level, target language

Interpreters directly execute the input language

- same front-end structure as a compiler
- then evaluate the annotated AST, or translate to intermediate code and evaluate that

Software engineering tools can resemble compilers

- same front-end structure as a compiler
- then:
  - pretty-print/reformat/colorize
  - analyze to compute relationships like declarations/uses, calls/callees, etc.
  - analyze to find potential bugs
  - aid in refactoring/restructuring/evolving programs

Engineering issues

Compilers are hard to design so that they are

- fast
- highly optimizing
- extensible & evolvable
- correct

Some parts of compilers can be automatically generated from specifications, e.g., scanners, parsers, & target code generators

- generated parts are fast & correct
- specifications are easily evolvable

(Some of my current research is on generating fast, correct optimizations from specifications.)

Need good management of software complexity

Lexical Analysis / Scanning

Purpose: turn character stream (input program) into **token stream**

- parser turns token stream into syntax tree

Token:
group of characters forming basic, atomic chunk of syntax;
a “word”

Whitespace:
characters between tokens that are ignored

Why separate lexical from syntactic analysis?

Separation of concerns / good design

- scanner:
  - handle grouping chars into tokens
  - ignore whitespace
  - handle I/O, machine dependencies
- parser:
  - handle grouping tokens into syntax trees

Restricted nature of scanning allows faster implementation

- scanning is time-consuming in many compilers

Complications

Most languages today are “free-form”

- layout doesn’t matter
- whitespace separates tokens

Alternatives:

- Fortran: line-oriented, whitespace doesn’t separate

```
        do 10 i = 1.100
          .. a loop ..
     10 continue
```

- Haskell: can use identation & layout to imply grouping

Most languages separate scanning and parsing

Alternative: C/C++/Java: ***type*** vs. ***identifier***

- parser wants scanner to distinguish names that are types from names that are variables
- but scanner doesn’t know how things declared -- that’s done during semantic analysis a.k.a. typechecking!

Lexemes, tokens, and patterns

Lexeme: group of characters that form a token

Token: class of lexemes that match a pattern

- token may have attributes, if more than one lexeme in token

Pattern: typically defined using a **regular expression**

- REs are simplest language class that’s powerful enough

Languages and language specifications

Alphabet: a finite set of characters/symbols

String: a finite, possibly empty sequence of characters in alphabet

Language: a (possibly empty or infinite) set of strings

Grammar: a finite specification of a set of strings

Language automaton:
a finite machine for accepting a set of strings and rejecting all others

A language can be specified by many different grammars and automata

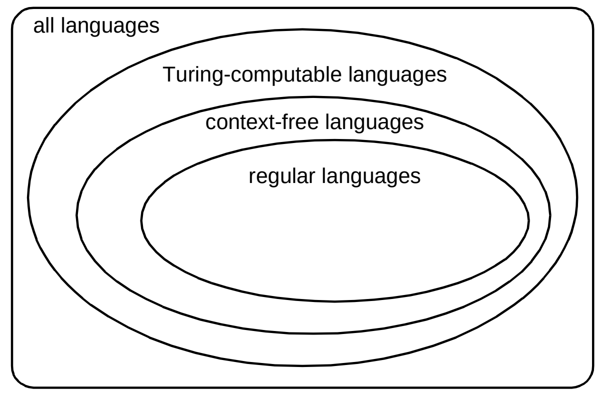
A grammar or automaton specifies only one language

Classes of languages

Regular languages can be specified by
regular expressions/grammars, finite-state automata (FSAs)

Context-free languages can be specified by
context-free grammars, push-down automata (PDAs)

Turing-computable languages can be specified by
general grammars, Turing machines



Syntax of regular expressions

Defined inductively

- base cases:
  - the empty string (ε or $\in$)
  - a symbol from the alphabet (e.g. **x**)
- inductive cases:
  - sequence of two RE's: E_1E_2
  - either of two RE's: $E_1 | E_2$
  - Kleene closure (zero or more occurrences) of a RE: E^*

Notes:

- can use parentheses for grouping
- precedence: * highest, sequence, | lowest
- whitespace insignificant

Notational conveniences

E^+ means 1 or more occurrences of *E*

E^k means k occurrences of *E*

[*E*] means 0 or 1 occurrence of *E* (optional *E*)

{*E*} means 0 or more occurrences of *E*

not*(*x*)** means any character in the alphabet but ***x

not(*E*) means any string of characters in the alphabet but those strings matching *E*

E_1-E_2 means any string matching E_1 except those matching E_2

No additional expressive power through these conveniences

Naming regular expressions

Can assign names to regular expressions

Can use the name of a RE in the definition of another RE

Examples:

```
letter      ::= a | b | ... | z
digit       ::= 0 | 1 | ... | 9
alphanum    ::= letter | digit
```

Grammar-like notation for named RE's: a regular grammar

Can reduce named RE's to plain RE by "macro expansion"

- no recursive definitions allowed,
  unlike full context-free grammars

Using regular expressions to specify tokens

Identifiers

```
ident       ::= letter (letter | digit)*
```

Integer constants

```
integer     ::= digit+
sign        ::= + | -
signed_int  ::= [sign] integer
```

Real number constants

```
real        ::= signed_int
                [fraction] [exponent]
fraction    ::= . digit+
exponent    ::= (E|e) signed_int
```


More token specifications

String and character constants

```
string      ::= " char* "
character   ::= ' char '

char        ::= not("|'|\) | escape
escape      ::= \("|'|\|n|r|t|v|b|a)
```

Whitespace

```
whitespace  ::= <space> | <tab> | <newline> |
                comment
comment     ::= /* not(*/)* */
```


Meta-rules

Can define a rule that a legal program is a sequence of tokens and whitespace

```
program ::= (token|whitespace)*
token ::= ident | integer | real | string | ...
```

But this doesn't say how to uniquely break up an input program into tokens -- it's highly ambiguous!

E.g. what tokens to make out of `hi2bob`?

- one identifier, `hi2bob`?
- three tokens, `hi 2 bob`?
- six tokens, each one character long?

The grammar states that it's legal, but not how tokens should be carved up from it

Apply extra rules to say how to break up string into sequence of tokens

- longest match wins
- reserved words take precedence over identifiers
- yield tokens, drop whitespace

RE specification of initial MiniJava lexical structure

```
Program     ::= (Token | Whitespace)*

Token       ::= ID | Integer | ReservedWord |
                Operator | Delimiter
ID          ::= Letter (Letter | Digit)*
Letter      ::= a | ... | z | A | ... | Z
Digit       ::= 0 | ... | 9
Integer     ::= Digit+
ReservedWord::= class | public | static |
                extends | void | int |
                boolean | if | else |
                while | return | true | false |
                this | new | String | main |
                System.out.println
Operator    ::= + | - | * | / | < | <= | >= |
                > | == | != | && | !
Delimiter   ::= ; | . | , | = |
                ( | ) | { | } | [ | ]

Whitespace  ::= <space> | <tab> | <newline>
```

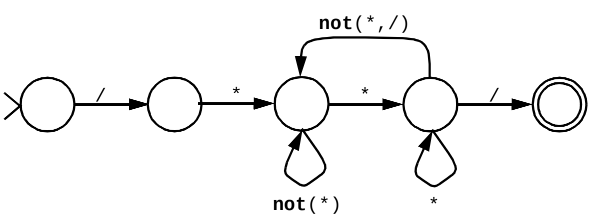

Building scanners from RE patterns

Convert RE specification into **finite state automaton (FSA)**

Convert FSA into scanner implementation
- by hand into collection of procedures
- mechanically into table-driven scanner

Finite state automata

An FSA has:
- a set of states
  - one marked the initial state
  - some marked final states
- a set of transitions from state to state
  - each transition labelled with a symbol from the alphabet or ε



Operate by reading symbols and taking transitions, beginning with the start state
- if no transition with a matching label is found, reject

When done with input, accept if in final state, reject otherwise

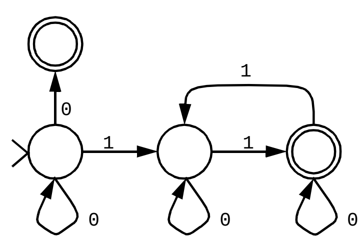
Determinism

FSA can be **deterministic** or **nondeterministic**

Deterministic: always know which way to go
- at most 1 arc leaving a state with particular symbol
- no ε arcs

Nondeterministic: may need to explore multiple paths, only choose right one later

Example:



NFAs vs. DFAs

A problem:

- RE's (e.g. specifications) map to NFA's easily
- Can write code from DFA easily

How to bridge the gap?
Can it be bridged?

A solution

Cool algorithm to translate any NFA into equivalent DFA!

- proves that NFAs aren't more expressive than DFAs

Plan:

1) Convert RE into NFA [they're equivalent]
2) Convert NFA into DFA
3) Convert DFA into code

Can be done by hand, or fully automatically

RE ⇒ NFA

Define by cases

ε

x

$E_1\ E_2$

$E_1 \mid E_2$

E^*

NFA ⇒ DFA

Problem: NFA can "choose" among alternative paths, while DFA must have only one path

Solution: **subset construction** of DFA

- each state in DFA represents *set of states in NFA*, all that the NFA might be in during its traversal

Subset construction algorithm

Given NFA with states and transitions

- label all NFA states uniquely

Create start state of DFA

- label it with the set of NFA states that can be reached by ε transitions (i.e. without consuming any input)

Process the start state

To process a DFA state *S* with label $\{s_1,..,s_N\}$:

For each symbol *x* in the alphabet:

- compute the set $\{t_1,..,t_M\}$ of NFA states reached from any of the NFA states in $\{s_1,..,s_N\}$ by an *x* transition followed by any number of ε transitions
- if $\{t_1,..,t_M\}$ not empty:
  - if an existing DFA state *T* has $\{t_1,..,t_M\}$ as a label, add a transition labeled *x* from *S* to *T*
  - otherwise create a new DFA state *T* labeled $\{t_1,..,t_M\}$, add a transition labeled *x* from *S* to *T*, and process *T*

A DFA state is final iff
at least one of the NFA states in its label is final

DFA ⇒ code

Option 1: implement scanner by hand using procedures

- one procedure for each token
- each procedure reads characters
- choices implemented using if & switch statements

Pros

- straightforward to write by hand
- fast

Cons

- a fair amount of tedious work
- may have subtle differences from language specification

DFA ⇒ code (cont.)

Option 2: use tool to generate table-driven scanner

- rows: states of DFA
- columns: input characters + EOF
- entries: action
  - go to new state
  - emit previous token, retry in start state
  - emit previous token, then done
  - done
  - report lexical error

Pros

- convenient for automatic generation
- exactly matches specification, if tool-generated

Cons

- “magic”
- table lookups may be slower than direct code
  - but switch statements get compiled into table lookups, so....
  - can translate table lookups into switch statements, if beneficial

Automatic scanner generation in MiniJava

We use the `jflex` tool to automatically create a scanner from a specification file, `Scanner/minijava.jflex`

(We use the CUP tool to automatically create a parser from a specification file, `Parser/minijava.cup`, which also generates all the code for the token classes used in the scanner, via the `Symbol` class.)

The MiniJava `Makefile` automatically rebuilds the scanner (or parser) whenever its specification file changes

Symbol class

Lexemes are represented as instances of class `Symbol`

```
class Symbol {
   int sym;        // which token class?
   Object value;   // any extra data for this lexeme
   ...
}
```

A different integer constant is defined for each token class, in the `sym` helper class

```
class sym {
   static int CLASS = 1;
   static int IDENTIFIER = 2;
   static int COMMA = 3;
   ...
}
```

Can use this in printing code for `Symbols`

- see `symbolToString` in `minijava.jflex`

Token declarations

Declare new token classes in `Parser/minijava.cup`, using `terminal` declarations

- include Java type if `Symbol` stores extra data

Examples:

```
/* reserved words: */
terminal CLASS, PUBLIC, STATIC, EXTENDS;
...
/* operators: */
terminal PLUS, MINUS, STAR, SLASH, EXCLAIM;
...
/* delimiters: */
terminal OPEN_PAREN, CLOSE_PAREN;
terminal EQUALS, SEMICOLON, COMMA, PERIOD;
...
/* tokens with values: */
terminal String IDENTIFIER;
terminal Integer INT_LITERAL;
```


`jflex` token specifications

Helper definitions for character classes and regular expressions

```
letter = [a-zA-Z]
eol = [\r\n]
```

(Simple) token definitions are of the form:

`regexp` `{` *`Java stmt`* `}`

`regexp` can be (at least):

- a string literal in double-quotes, e.g. `"class"`, `"<="`
- a reference to a named helper, in braces, e.g. `{letter}`
- a character list or range, in square brackets, e.g. `[a-zA-Z]`
- a negated character list or range, e.g. `[^\r\n]`
- `.` (which matches any single character)
- *regexp regexp*, *regexp* | *regexp*, *regexp**, *regexp*+, *regexp*?, (*regexp*)

`Java stmt` (the accept action) is typically:

- `return symbol(sym.CLASS);` for a simple token
- `return symbol(sym.CLASS, yytext());` for a token with extra data based on the lexeme string `yytext()`
- empty for whitespace