CS3300 - Compiler Design

Introduction

V. Krishna Nandivada

IIT Madras

Academic Formalities

- Written assignments = 5+5 marks.
- Quiz 1 = 10 marks, Quiz 2 = 10, Final = 40 marks.
- Programming assignments: Six assignments. Total 40 marks.
- Extra marks
  - During the lecture time - individuals can get additional 5 marks.
  - How? - Ask a good question, answer a chosen question, make a good point! Take 0.5 marks each. Max one mark per day per person.
- Attendance requirement – as per institute norms.
  - If you come to the class after 5 minutes - don't.
  - Proxy attendance - is not a help; actually a disservice.
- Plagiarism - A good word to know. A bad act to own.
  - Students Welfare and Disciplinary committee.

Contact (Anytime) :
Instructor: Krishna, Email: nvk@iitm.ac.in, Office: SSB 406.
Details about the course: http://www.cse.iitm.ac.in/∼krishna/cs3300/

What, When and Why of Compilers

- **What**:
  - A compiler is a program that can read a program in one language and translates it into an equivalent program in another language.
- **When**
  - 1952, by Grace Hopper for A-0.
  - 1957, Fortran compiler by John Backus and team.
- **Why? Study?**
  - It is good to know how the food (you eat) is cooked.
  - A programming language is an artificial language designed to communicate instructions to a machine, particularly a computer.
  - For a computer to execute programs written in these languages, these programs need to be translated to a form in which it can be executed by the computer.

Images of the day

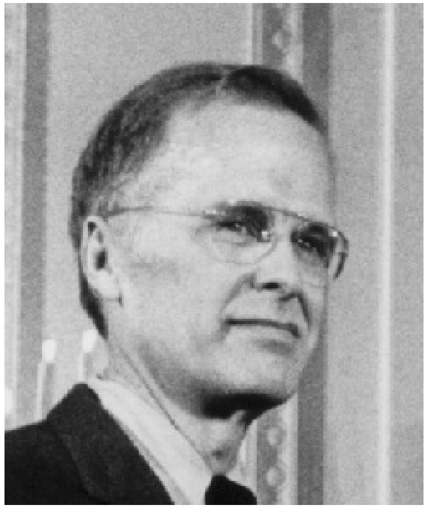


Figure: Grace Hopper and John Backus

Compilers – A “Sangam”

Compiler construction is a microcosm of computer science

- **Artificial Intelligence** greedy algorithms, learning algorithms, . . .
- **Algo** graph algorithms, union-find, dynamic programming, . . .
- **theory** DFAs for scanning, parser generators, lattice theory, . . .
- **systems** allocation, locality, layout, synchronization, . . .
- **architecture** pipeline management, hierarchy management, instruction set use, . . .
- **optimizations** Operational research, load balancing, scheduling, . . .

Inside a compiler, all these and many more come together. Has probably the healthiest mix of theory and practise.

Mutual expectations

For the class to be a mutually learning experience:

- What will be required from the students?
  - An open mind to learn.
  - Curiosity to know the basics.
  - Explore their own thought process.
  - Help each other to learn and appreciate the concepts.
  - Honesty and hard work.
  - Leave the fear of marks/grades.
- What are the students expectations?

Course outline

A rough outline (we may not strictly stick to this).

- Overview of Compilers
- Regular Expressions and Context Free Grammars (glance)
- Lexical Analysis and Parsing
- Type checking
- Intermediate Code Generation
- Register Allocation
- Code Generation
- Overview of advanced topics.

Goal of the course: At the end of the course, students will have a fair understanding of some standard passes in a general purpose compiler. Students will have hands on experience on implementing a compiler for a subset of Java.

Your friends: Languages and Tools

Start exploring

- C and Java - familiarity a must - Use eclipse to save you valuable coding and debugging cycles.
- Flex, Bison, JavaCC, JTB – tools you will learn to use.
- Make / Ant / Scripts – recommended toolkit.
- Find the course webpage: http://www.cse.iitm.ac.in/∼krishna/cs3300/

Get set. Ready steady go!

Acknowledgement

These slides borrow liberal portions of text verbatim from Antony L. Hosking @ Purdue, Jens Palsberg @ UCLA, and the Dragon book.

A common confusion: Compilers and Interpreters

- What is a compiler?
  - a program that translates an executable program in one language into an executable program in another language
  - we expect the program produced by the compiler to be better, in some way, than the original.
- What is an interpreter?
  - a program that reads an executable program and produces the results of running that program
  - usually, this involves executing the source program in some fashion

  This course deals mainly with compilers
  Many of the same issues arise in interpreter
- A common (mis?) statement – XYZ is an interpreted (or compiled) languaged.

Compilers – A closed area?

“Optimization for scalar machines was solved years ago”

Machines have changed drastically in the last 20 years

Changes in architecture $\Rightarrow$ changes in compilers

- new features pose new problems
- changing costs lead to different concerns
- old solutions need re-engineering

Changes in compilers should prompt changes in architecture

- New languages and features

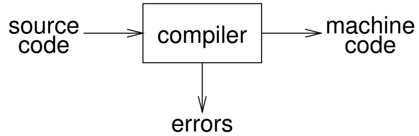
Expectations

What qualities are important in a compiler?

1. Correct code
2. Output runs fast
3. Compiler runs fast
4. Compile time proportional to program size
5. Support for separate compilation
6. Good diagnostics for syntax errors
7. Works well with the debugger
8. Good diagnostics for flow anomalies
9. Cross language calls
10. Consistent, predictable optimization

Each of these shapes your expectations about this course

Abstract view

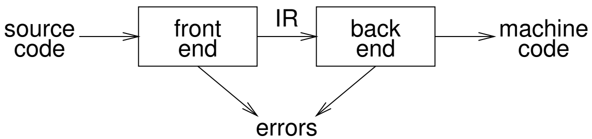


Implications:

- recognize legal (and illegal) programs
- generate correct code
- manage storage of all variables and code
- agreement on format for object (or assembly) code

Big step up from assembler — higher level notations

Traditional two pass compiler

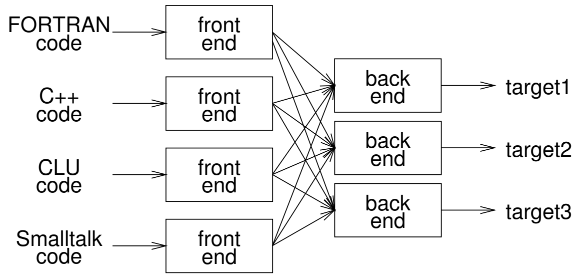


Implications:

- intermediate representation (IR). Why do we need it?
- front end maps legal code into IR
- back end maps IR onto target machine
- simplify retargeting
- allows multiple front ends
- multiple passes $\Rightarrow$ better code

A rough statement: Most of the problems in the Front-end are simpler (polynomial time solution exists).
Most of the problems in the Back-end are harder (many problems are NP-complete in nature).
Our focus: Mainly front end and little bit of back end.

A Clarification:



Can we build $n \times m$ compilers with $n + m$ components?

- must encode all the knowledge in each front end
- must represent all the features in one IR
- must handle all the features in each back end

Limited success with low-level IRs

Phases inside the compiler

character-stream
↓
Lexical Analyzer
↓ token-stream
Syntax Analyzer
↓ syntax-tree
Semantic Analyzer
↓ syntax-tree
Intermediate Code Generator
↓ intermediate-representation
Machine-Independent Opt
↓ intermediate-representation
Code Generation
↓ target-machine-code (IR)
Machine-dependent Opt
↓ target-machine-code

Front end responsibilities:

- Recognize syntactically legal code; report errors.
- Recognize semantically legal code; report errors.
- Produce IR.

Back end responsibilities:

- Optimizations, code generation.

Our target

- five out of seven phases.
- glance over optimizations – attend the graduate course, if interested.

Lexical analysis

- Also known as scanning.
- Reads a stream of characters and groups them into meaningful sequences, called lexems.
- Eliminates white space
- For each lexeme, the scanner produces an output of the form: ⟨token-type, attribute-values⟩
- Example token-types: identifier, number, string, operator and ...
- Example attribute-types: token index, token-value, line and column number and ...
- Example scanning:
  - `position = initial + rate * 60`
  - For a typical language like C/Java the following lexemes and their values can be identified:

lexeme	token
`position`	⟨id, position⟩
`=`	⟨op, =⟩
`initial`	⟨id, initial⟩

lexeme	token
`+`	⟨op, +⟩
`rate`	⟨id, rate⟩
`*`	⟨op, *⟩
`60`	⟨num, 60⟩

Specifying patterns

Q: How to specify patterns for the scanner?

Examples:

- white space

  <ws> ::= <ws> ' '
  | <ws> '\t'
  | ' '
  | '\t'

- keywords and operators
  specified as literal patterns: `do`, `end`

Specifying patterns

A scanner must recognize the units of syntax

- identifiers
  alphabetic followed by k alphanumerics (_, $, &, ...)
- numbers
  - integers: 0 or digit from 1-9 followed by digits from 0-9
  - decimals: integer |'.'| digits from 0-9
  - reals: (integer or decimal) |'E'| (+ or -) digits from 0-9
  - complex: |'('| real |','| real |')'—

We need a powerful notation to specify these patterns

Regular Expressions

Patterns are often specified as regular languages

Notations used to describe a regular language (or a regular set) include both regular expressions and regular grammars

Regular expressions (over an alphabet Σ):

1. ε is a RE denoting the set $\{\varepsilon\}$
2. if $a \in \Sigma$, then a is a RE denoting $\{a\}$
3. if r and s are REs, denoting $L(r)$ and $L(s)$, then:
   - (r) is a RE denoting $L(r)$
   - $(r) \mid (s)$ is a RE denoting $L(r) \bigcup L(s)$
   - $(r)(s)$ is a RE denoting $L(r)L(s)$
   - $(r)^*$ is a RE denoting $L(r)^*$

Examples of Regular Expressions

- identifier
  letter $\rightarrow (a \mid b \mid c \mid ... \mid z \mid A \mid B \mid C \mid ... \mid Z)$
  digit $\rightarrow (0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)$
  id $\rightarrow$ letter (letter | digit)*
- numbers
  integer $\rightarrow (+ \mid - \mid \varepsilon)\ (0 \mid (1 \mid 2 \mid 3 \mid ... \mid 9)$ digit$^*)$
  decimal $\rightarrow$ integer . (digit)*
  real $\rightarrow$ (integer | decimal) E $(+ \mid -)$ digit*
  complex $\rightarrow$ '(' real , real ')'

Most tokens can be described with REs

We can use REs to build scanners automatically

Generic examples of REs

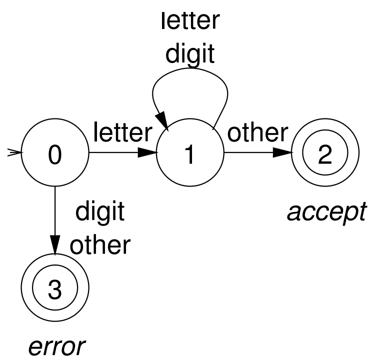
Let $\Sigma = \{a, b\}$

- $a|b$ denotes $\{a, b\}$
- $(a|b)(a|b)$ denotes $\{aa, ab, ba, bb\}$
  i.e., $(a|b)(a|b) = aa|ab|ba|bb$
- $a*$ denotes $\{\varepsilon, a, aa, aaa, \ldots\}$
- $(a|b)*$ denotes the set of all strings of a's and b's (including ε)
  i.e., $(a|b)* = (a*b*)*$
- $a|a*b$ denotes $\{a, b, ab, aab, aaab, aaaab, \ldots\}$

Recognizers

From a regular expression we can construct a
deterministic finite automaton (*DFA*)

Recognizer for identifier:



Code for the recognizer

Given an automata, can we write a recognizer for a token?

```
ch=nextChar();
state=0; // initial state
done=false;
tokenVal=""// empty
while (not done) {
 class=charClass[ch];
 state=
   nextState[class,state];
 switch(state) {
  case 1:
    tokenVal=tokenVal+ch;
    ch=nextChar();
    break;
case 2: // accept state
      tokenType=id;
      done = true;
      break;
    case 3: // error
      tokenType=error;
      done=true;
      break;
  } // end switch
} // end while
return tokenType;
```


Tables for the recognizer

Two tables control the recognizer

`charClass`:

	$a-z$	$A-Z$	$0-9$	other
value	letter	letter	digit	other

`nextState`:

class	0	1	2	3
letter	1	1	—	—
digit	3	1	—	—
other	3	2	—	—

To change languages, we can just change tables

So what is hard?

Language features that can cause problems:

reserved words
PL/I had no reserved words
`if then then then = else; else else = then;`

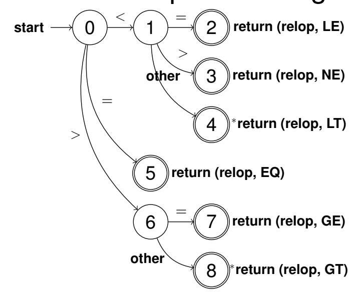
significant blanks
FORTRAN and Algol68 ignore blanks
`do 10 i = 1,25`
`do 10 i = 1.25`

string constants
special characters in strings
`newline`, `tab`, `quote`, `comment delimiter`

finite closures
some languages limit identifier lengths
adds states to count length
FORTRAN 66 → 6 characters

Considerations when building lexical analyzer

- How to combine multiple DFAs?
  - Try all (in parallel?), take the longest.
- Some of the patterns may have common prefixes. e.g. <, <=, <>
- Create a transition diagram.



- Reserved words: example `then`, `thenVar`
  - Identify as an identifier and if the value matches a reserved word, change their "type".
  - Let it be identified as both reserved word and identifier. Higher priority to reserved words.

Error recovery

- It is hard to tell (without the aid of other components), if there is a source code error.
- For example:
  `fi (a = f(x))`
  If `fi` a misspelling for "`if`", or a function identifier?
- Since `fi` is a valid lexeme for the token `id`, the lexer must return the token ⟨id, fi⟩.
- A later phase (parser or semantic analyzer) may be able to catch the error.

Recovery (if the lexer is unable to proceed, that is):

- Panic and stop!
- Delete one character!
- Many other one character related fixes (examples?)

Automatic construction

Scanner generators automatically construct code from RE-like descriptions

- construct a DFA
- use state minimization techniques
- emit code for the scanner
  (table driven or direct code)

A key issue in automation is an interface to the parser

`lex/flex` is a scanner generator

- Takes a specification of all the patterns as a RE.
- emits C code for scanner
- provides macro definitions for each token
  (used in the parser)

Limits of regular languages

Not all languages are regular
One cannot construct DFAs to recognize these languages:

- $L = \{p^k q^k\}$
- $L = \{wcw^r \mid w \in \Sigma*\}$

Note: neither of these is a regular expression!
(DFAs cannot count!)
But, this is a little subtle. One can construct DFAs for:

- alternating 0's and 1's
  $(\varepsilon \mid 1)(01)*(\varepsilon \mid 0)$
- sets of pairs of 0's and 1's
  $(01 \mid 10)+$