CS591-5 - Selected topics in Compiler Design
Introduction

V. Krishna Nandivada

IIT Madras

Academic Formalities

- Written assignments = 10 marks.
- Final = 40 marks.
- Programming assignment: One assignment (optional).
- 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. Non compliance will lead to 'W' grade.
  - 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: A6-04.
Course page:
`http://www.cse.iitm.ac.in/~krishna/cs591-5/`

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.

Course outline

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

- Overview of Compilers
- Lexical Analysis and Parsing (overview)
- Intermediate Code (three address codes)
- Data flow analysis
- Constant propagation

Your friends: Languages and Tools

Start exploring

- C and Java - familiarity a must - Use eclipse to save you valuable coding and debugging cycles.
- Find the course webpage: `http://www.cse.iitm.ac.in/~krishna/cs591-5/`

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 ⇒ 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

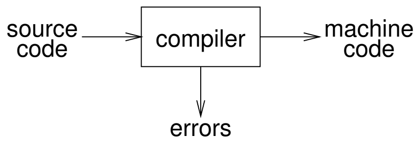
Expectations

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

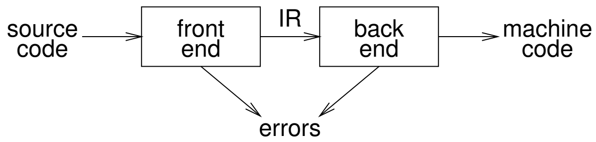


Implications:

- 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

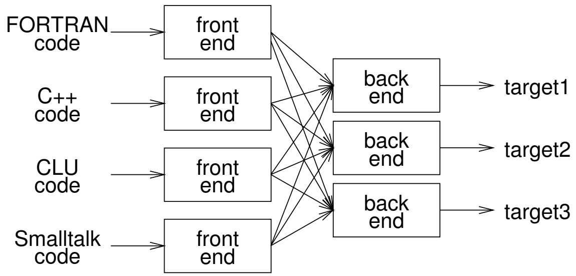


Implications:

- 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:

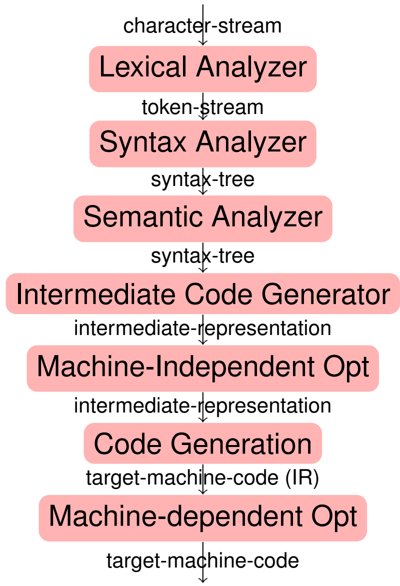


Can we build $n \times m$ compilers with $n+m$ components?

- 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



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

- three out of seven phases.
- briefly touch upon the rest. Based on the need.

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 $\rightarrow (a \mid b \mid c \mid \ldots \mid z \mid A \mid B \mid C \mid \ldots \mid Z)$
  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 $\rightarrow (+ \mid - \mid \varepsilon)\ (0 \mid (1 \mid 2 \mid 3 \mid \ldots \mid 9)$ 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

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:

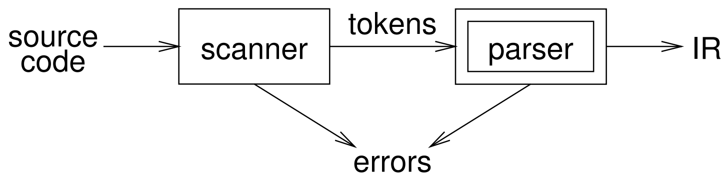
- $L = \{p^k q^k\}$
- $L = \{wcw^r \mid w \in \Sigma*\}$

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
  $(01 \mid 10)+$

Q: What do the above languages denote?

The role of the parser



A parser

- performs context-free syntax analysis
- guides context-sensitive analysis
- constructs an intermediate representation
- produces meaningful error messages
- attempts error correction

Syntax analysis

Grammars are often written in Backus-Naur form (BNF).
Example:

1	$\langle\text{goal}\rangle$	$::=$	$\langle\text{expr}\rangle$
2	$\langle\text{expr}\rangle$	$::=$	$\langle\text{expr}\rangle\langle\text{op}\rangle\langle\text{expr}\rangle$
3		$\mid$	`num`
4		$\mid$	`id`
5	$\langle\text{op}\rangle$	$::=$	$+$
6		$\mid$	$-$
7		$\mid$	$*$
8		$\mid$	$/$

This describes simple expressions over numbers and identifiers.
In a BNF for a grammar, we represent

1. non-terminals with angle brackets or capital letters
2. terminals with `typewriter` font or <u>underline</u>
3. productions as in the example

Derivations

We can view the productions of a CFG as rewriting rules.
Using our example CFG (for $\mathtt{x} + \mathtt{2} * \mathtt{y}$):

$$\begin{array}{rcl}
\langle\text{goal}\rangle & \Rightarrow & \langle\text{expr}\rangle \\
& \Rightarrow & \langle\text{expr}\rangle\langle\text{op}\rangle\langle\text{expr}\rangle \\
& \Rightarrow & \langle\text{id,x}\rangle\langle\text{op}\rangle\langle\text{expr}\rangle \\
& \Rightarrow & \langle\text{id,x}\rangle + \langle\text{expr}\rangle \\
& \Rightarrow & \langle\text{id,x}\rangle + \langle\text{expr}\rangle\langle\text{op}\rangle\langle\text{expr}\rangle \\
& \Rightarrow & \langle\text{id,x}\rangle + \langle\text{num,2}\rangle\langle\text{op}\rangle\langle\text{expr}\rangle \\
& \Rightarrow & \langle\text{id,x}\rangle + \langle\text{num,2}\rangle * \langle\text{expr}\rangle \\
& \Rightarrow & \langle\text{id,x}\rangle + \langle\text{num,2}\rangle * \langle\text{id,y}\rangle
\end{array}$$

We have derived the sentence $\mathtt{x} + \mathtt{2} * \mathtt{y}$.
We denote this $\langle\text{goal}\rangle \rightarrow^* \mathtt{id} + \mathtt{num} * \mathtt{id}$.
Such a sequence of rewrites is a <u>derivation</u> or a <u>parse</u>.
The process of discovering a derivation is called <u>parsing</u>.
Parse Tree is generated.

Scanning vs. parsing

<u>Where do we draw the line?</u>

$$\begin{array}{lcl}
term & ::= & [\mathtt{a}-\mathtt{zA}-\mathtt{z}]([\mathtt{a}-\mathtt{zA}-\mathtt{z}] \mid [0-9])^* \\
& \mid & 0 \mid [1-9][0-9]^* \\
op & ::= & + \mid - \mid * \mid / \\
expr & ::= & (term\ op)^* term
\end{array}$$

Regular expressions are used to classify:

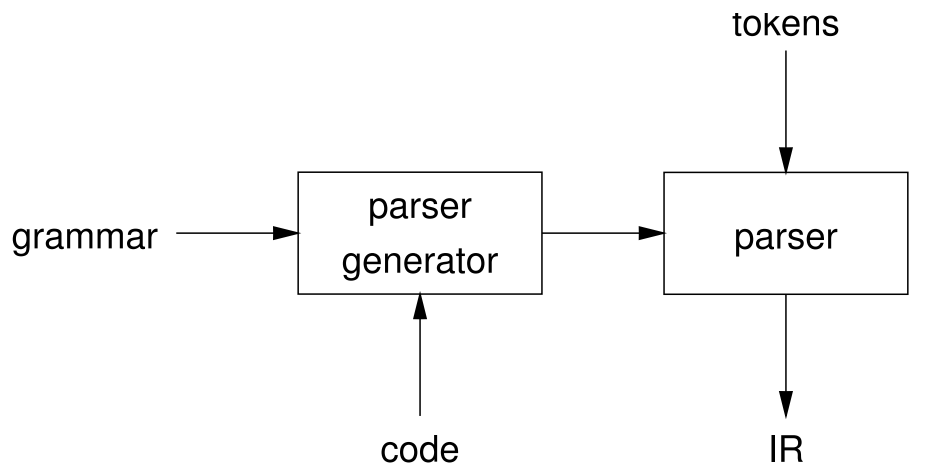
- identifiers, numbers, keywords
- REs are more concise and simpler for tokens than a grammar
- more efficient scanners can be built from REs (DFAs) than grammars

Context-free grammars are used to count:

- brackets: `()`, `begin...end`, `if...then...else`
- imparting structure: expressions

<u>Syntactic analysis is complicated enough: grammar for C has around 200 productions. Factoring out lexical analysis as a separate phase makes compiler more manageable.</u>

Parsing: the big picture



<u>Automatic Generation:</u>
<u>Lexer: From REs</u>
<u>Parser: From CFGs</u>

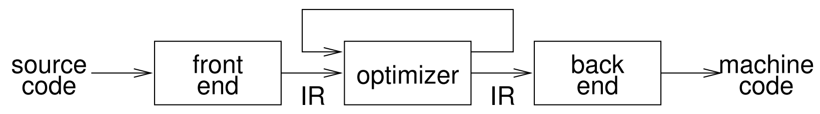
Intermediate representations

<u>Why use an intermediate representation?</u>

1. break the compiler into manageable pieces
   – good software engineering technique
2. simplifies retargeting to new host
   – isolates back end from front end
3. simplifies handling of “poly-architecture” problem
   – m lang’s, n targets $\Rightarrow m+n$ components (<u>myth</u>)
4. enables machine-independent optimization
   – general techniques, multiple passes

<u>An intermediate representation is a compile-time data structure</u>

Intermediate representations



Generally speaking:

- front end produces IR
- optimizer transforms that representation into an equivalent program that may run more efficiently
- back end transforms IR into native code for the target machine

Intermediate representations - properties

<u>Important IR Properties</u>

- ease of generation
- ease of manipulation
- cost of manipulation
- level of abstraction
- freedom of expression
- size of typical procedure

Subtle design decisions in the IR have far reaching effects on the speed and effectiveness of the compiler.
Level of exposed detail is a crucial consideration.

IR design issues

- Is the chosen IR appropriate for the (analysis/ optimization/ transformation) passes under consideration?
- What is the IR level: close to language/machine.
- Multiple IRs in a compiler: for example, High, Medium and Low

```
                  t1 = j + 2        r1 = [fp-4]
x = a[i,j+2]      t2 = i * 2        r2 = r1 + 2
                  t3 = t1 + t2      r3 = [fp-8]
                  t4 = 4 * t3       r4 = r3 * 20
                  t5 = addr a       r5 = r4 + r2
                  t6 = t5 + t4      r6 = 4 * r5
                  x = *t6           r7 = fp - 216
                                    f1 = [r7+r6]
```

- In reality, the variables etc are also only pointers to other data structures.

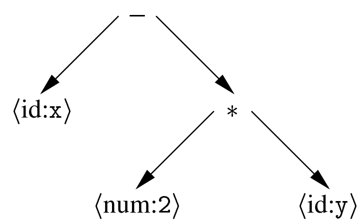
Intermediate representations

Representations talked about in the literature include:

- abstract syntax trees (AST)
- linear (operator) form of tree
- directed acyclic graphs (DAG)
- control flow graphs
- program dependence graphs
- static single assignment form
- 3-address code
- hybrid combinations

Abstract syntax tree

An abstract syntax tree (AST) is the procedure's parse tree with the nodes for most non-terminal symbols removed.

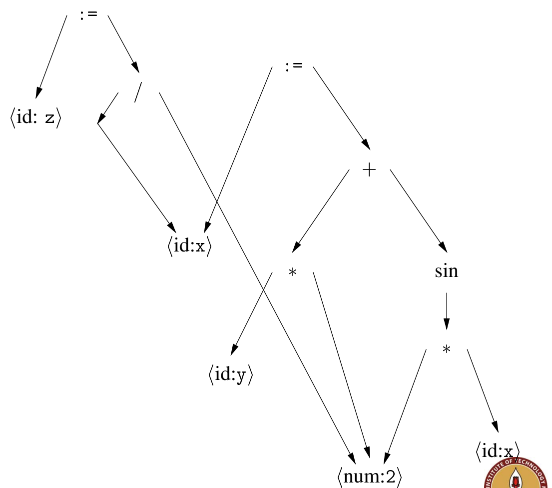


This represents "$\mathtt{x} - 2 * \mathtt{y}$".
For ease of manipulation, can use a linearized (operator) form of the tree.
e.g., in postfix form: `x 2 y` $* -$

Directed acyclic graph

A directed acyclic graph (DAG) is an AST with a unique node for each value.

```
x := 2 * y + sin(2*x)
z := x / 2
```



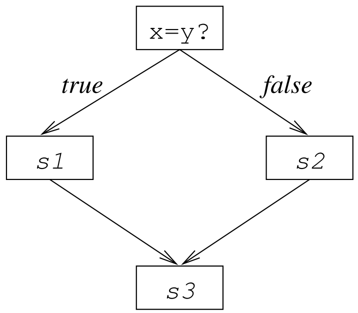
Q: What to do for matching names present across different functions?

Control flow graph

The control flow graph (CFG) models the transfers of control in the procedure

- nodes in the graph are basic blocks
  straight-line blocks of code
- edges in the graph represent control flow
  loops, if-then-else, case, goto

```
if (x=y) then
   s1
else
   s2
s3
```



3-address code

- At most one operator on the right side of an instruction.
- 3-address code can mean a variety of representations.
- In general, it allows statements of the form:
  `x ← y op z`
  with a single operator and, at most, three names.
  Simpler form of expression:
  `x - 2 * y`
  becomes
  `t1 ← 2 * y`
  `t2 ← x - t1`

Advantages

- compact form (direct naming)
- names for intermediate values

Can include forms of prefix or postfix code

3-address code: Addresses

Three-address code is built from two concepts: addresses and instructions.

- An address can be
  - A name: source variable program name or pointer to the Symbol Table name.
  - A constant: Constants in the program.
  - Compiler generated temporary.

3-address code

Typical instructions types include:

1. assignments `x ← y op z`
2. assignments `x ← op y`
3. assignments `x ← y[i]`
4. assignments `x ← y`
5. branches `goto L`
6. conditional branches
   `if x goto L`
7. procedure calls
   `param` x_1, `param` x_2, ...`param` x_n
   and
   `call p, n`
8. address and pointer assignments

How to translate:

`if (x < y) S1 else S2`

?

Advice

- Many kinds of IR are used in practice.
- There is no widespread agreement on this subject.
- A compiler may need several different IRs
- Choose IR with right level of detail
- Keep manipulation costs in mind

Gap between HLL and IR

Gap between HLL and IR

- High level languages may allow complexities that are not allowed in IR (such as expressions with multiple operators).
- High level languages have many syntactic constructs, not present in the IR (such as if-then-else or loops)

Challenges in translation:

- Deep nesting of constructs.
- Recursive grammars.
- We need a systematic approach to IR generation.

Goal:

- A HLL to IR translator.
- Input: A program in HLL.
- Output: A program in IR (may be an AST or program text)

Translating expressions

```
S -> id = E;
E -> E1 + E2
   | - E1
   | (E1)
   | id
```


Translating flow-of-control statements

```
P -> S
S -> assign
     | S1; S2
     | if (expr) S1
     | if (expr) S2 else S3
     | while (expr) S1;
```


Translating boolean expressions

```
B -> B1 || B2
     | B1 && B2
     | !B
     | E1 rel E2
     | true
     | false
```

Self reading

- Translating Array dereference.
- Translating Switch, continue, break
- Translating object dereferences (advanced)
- Translating Exceptions (advanced++).