CS6013 - Modern Compilers: Theory and Practise

Introduction

V. Krishna Nandivada

IIT Madras

Academic Formalities

- Written assignment = 5 marks.
- Programming assignments = 40 marks.
- Midterm = 25 marks, Final = 30 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. Non compliance will lead to ‘W’ grade.
  - Proxy attendance - is not a help; actually a disservice.
- Plagiarism - A good word to know. A bad act to own.
  - Will be automatically referred to the institute welfare and disciplinary committee.

Contact (Anytime) :
Instructor: Krishna, Email: nvk@cse.iitm.ac.in, Office: BSB 352.
TA : Jyothi Vedurada, Email: vjyothi@cse

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

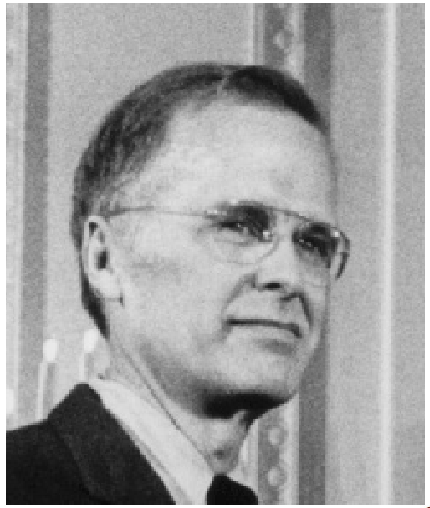


Figure: Grace Hopper and John Backus

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
- Overview of lexical analysis and parsing.
- Semantic analysis (aka type checking)
- Intermediate code generation
- Data flow analysis
- Constant propagation
- Static Single Assignment and Optimizations.
- Loop optimizations
- Liveness analysis
- Register Allocation
- Bitwidth aware register allocation
- Code Generation
- Overview of advanced topics.

Your friends: Languages and Tools

Start exploring

- Java - familiarity a must - Use eclipse to save you valuable coding and debugging cycles.
- JavaCC, JTB – tools you will learn to use.
- Make Ant Scripts – recommended toolkit.
- Find the course webpage: http://www.cse.iitm.ac.in/∼krishna/cs6013/

Get set. Ready steady go!

Acknowledgement

These frames borrow liberal portions of text verbatim from Antony L. Hosking @ Purdue and Jens Palsberg @ UCLA.

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

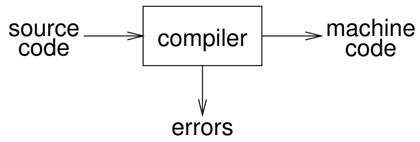
Expectations

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

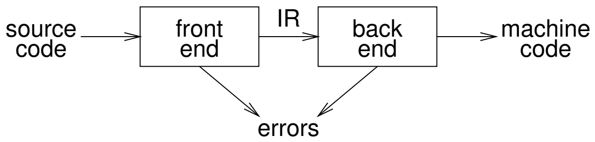


Implications:

- 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

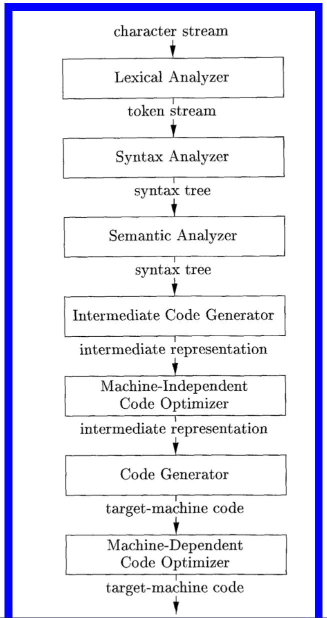


Implications:

- intermediate representation (IR).
- front end maps legal code into IR
- back end maps IR onto target machine
- simplify retargeting
- allows multiple front ends
- multiple passes ⇒ 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 back end (95%) and little bit of front end (5%).

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

- five out of seven phases.
- glance over lexical and syntax analysis – read yourself or attend the undergraduate course, if interested.

Lexical analysis

- Also known as scanning.
- Reads a stream of characters and groups them into meaningful sequences, called lexems.

A scanner must recognize the units of syntax

Q: How to specify patterns for the scanner?

Examples:

- white space

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

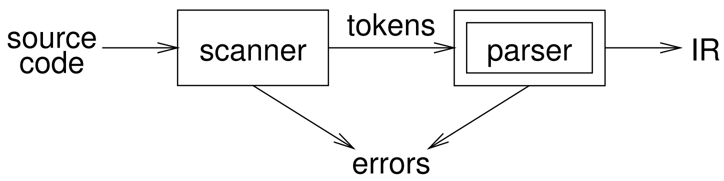
- keywords and operators
  specified as literal patterns: `do`, `end`

More complex syntax

- identifiers
  alphabet 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.
There are mature tools (e.g., flex) that generate lexical token generators (or scanners) from a given specification of tokens (a.k.a. sequence of regular expressions).

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 by using a CFG

Context-free syntax is specified with a context-free grammar.
Formally, a CFG G is a 4-tuple (V_t, V_n, S, P), where:

- V_t is the set of terminal symbols in the grammar. For our purposes, V_t is the set of tokens returned by the scanner.
- V_n, the nonterminals, is a set of syntactic variables that denote sets of (sub)strings occurring in the language. These are used to impose a structure on the grammar.
- S is a distinguished nonterminal $(S \in V_n)$ denoting the entire set of strings in $L(G)$. This is sometimes called a goal symbol.
- P is a finite set of productions specifying how terminals and non-terminals can be combined to form strings in the language. Each production must have a single non-terminal on its left hand side.

The set $V = V_t \cup V_n$ is called the vocabulary of G

Notation and terminology

- $a, b, c, \ldots \in V_t$
- $A, B, C, \ldots \in V_n$
- $U, V, W, \ldots \in V$
- $\alpha, \beta, \gamma, \ldots \in V*$
- $u, v, w, \ldots \in V_t*$

If $A \rightarrow \gamma$ then $\alpha A \beta \Rightarrow \alpha\gamma\beta$ is a single-step derivation using $A \rightarrow \gamma$

Similarly, $\rightarrow^*$ and $\Rightarrow^+$ denote derivations of ≥ 0 and ≥ 1 steps

If $S \rightarrow^* \beta$ then β is said to be a sentential form of G

$L(G) = \{w \in V_t* \mid S \Rightarrow^+ w\}$, $w \in L(G)$ is called a sentence of G

Note, $L(G) = \{\beta \in V* \mid S \rightarrow^* \beta\} \cap V_t*$

Derivations

We can view the productions of a CFG as rewriting rules.
Using an example CFG:

1	$\langle$goal$\rangle$	::=	$\langle$expr$\rangle$
2	$\langle$expr$\rangle$	::=	$\langle$expr$\rangle + \langle$term$\rangle$
3		|	$\langle$expr$\rangle - \langle$term$\rangle$
4		|	$\langle$term$\rangle$
5	$\langle$term$\rangle$	::=	$\langle$term$\rangle * \langle$factor$\rangle$
6		|	$\langle$term$\rangle / \langle$factor$\rangle$
7		|	$\langle$factor$\rangle$
8	$\langle$factor$\rangle$	::=	`num`
9		|	`id`

Deriving the derivation

Now, for the string $x + 2 * y$:

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

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

Different ways of parsing: Top-down Vs Bottom-up

Top-down parsers

- start at the root of derivation tree and fill in
- picks a production and tries to match the input
- may require backtracking
- some grammars are backtrack-free (*predictive*)

Bottom-up parsers

- start at the leaves and fill in
- start in a state valid for legal first tokens
- as input is consumed, change state to encode possibilities (recognize valid prefixes)
- use a stack to store both state and sentential forms

Top-down parsing

A top-down parser starts with the root of the parse tree, labelled with the start or goal symbol of the grammar.
To build a parse, it repeats the following steps until the fringe of the parse tree matches the input string

1. At a node labelled A, select a production $A \rightarrow \alpha$ and construct the appropriate child for each symbol of α
2. When a terminal is added to the fringe that doesn't match the input string, backtrack
3. Find next node to be expanded (must have a label in V_n)

The key is selecting the right production in step 1.

If the parser makes a wrong step, the "derivation" process does not terminate.
Why is it bad?

How much lookahead is needed?

We saw that top-down parsers may need to backtrack when they select the wrong production
Do we need arbitrary lookahead to parse CFGs?

- in general, yes
- use the Earley or Cocke-Younger, Kasami algorithms

Fortunately

- large subclasses of CFGs can be parsed with limited lookahead
- most programming language constructs can be expressed in a grammar that falls in these subclasses

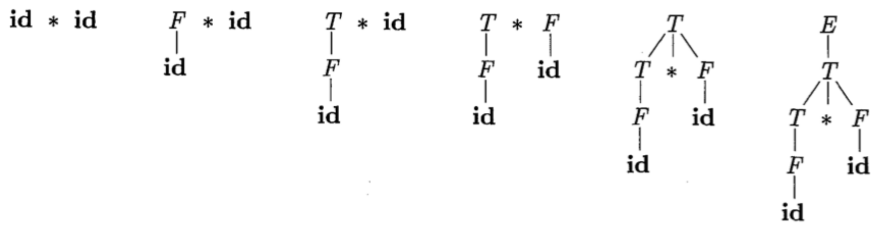
Among the interesting subclasses are:

LL(1): **l**eft to right scan, **l**eft-most derivation, **1**-token lookahead; and
LR(1): **l**eft to right scan, reversed **r**ight-most derivation, **1**-token lookahead

Bottom-up parsing

Goal:

Given an input string w and a grammar G, construct a parse tree by starting at the leaves and working to the root.



Reductions Vs Derivations

Reduction:

- At each reduction step, a specific substring matching the body of a production is replaced by the non-terminal at the head of the production.

Key decisions

- When to reduce?
- What production rule to apply?

Reduction Vs Derivations

- Recall: In derivation: a non-terminal in a sentential form is replaced by the body of one of its productions.
- A reduction is reverse of a step in derivation.
- Bottom-up parsing is the process of "reducing" a string w to the start symbol.
- Goal of bottum-up parsing: build derivation tree in reverse.

Parsing review

- Recursive descent
  A hand coded recursive descent parser directly encodes a grammar (typically an LL(1) grammar) into a series of mutually recursive procedures. It has most of the linguistic limitations of LL(1).
- LL(k)
  An LL(k) parser must be able to recognize the use of a production after seeing only the first k symbols of its right hand side.
- LR(k)
  An LR(k) parser must be able to recognize the occurrence of the right hand side of a production after having seen all that is derived from that right hand side with k symbols of lookahead.

There are mature tools (e.g., bison) that generate parsers from a given specification of syntax (a.k.a. grammar).

Closing remarks - parsing

- Overview of Parsing.
- Error checking.
- LR parsing.

Reading:

- Ch 1, 3, 4 from the Dragon book.

Announcement:

- Assignment 1 is out. Due in around 10 days.
- Next class: ?