

CMSC 430

CMSC 430—“Theory of Language Translation”

Topics in the design of programming language translators, including scanning, parsing, error recovery, code generation, and code improvement.

Prerequisite: CMSC 330

Important facts:

Name: Chau-Wen Tseng

Email: tseng@cs.umd.edu

Office: A.V. Williams 4135

Hours: Tue & Thu 3:30–4:30pm

TA: Nan Wang

Email: nwang@cs.umd.edu

Office: A.V. Williams 4176

Hours: Wed 1–3pm

Class Email: cmsc430@cs.umd.edu

Class URL: [csd.cmsc430 news.umd.edu](http://www.cs.umd.edu/news.umd.edu)

Newsgroup: <http://www.cs.umd.edu/~tseng/>

Textbook is *Modern Compiler Implementation* by Andrew Appel

Course Overview

Basis for grades:

- 20% mid-term exam
- 30% final exam
- 50% programming projects

Programming Projects

- scanner construction (REs to minimal DFAs)
- scanner/parser using JLex and CUP
- simple type checker
- Java byte code generation
- advanced code generation, optimizations

Policies

- no collaboration (code sharing) allowed
- 1-week late policy, no incompletes

Lecture notes

- all transparencies are on the Web
- you should still take notes, read textbook

Compiler Overview

What is a compiler?

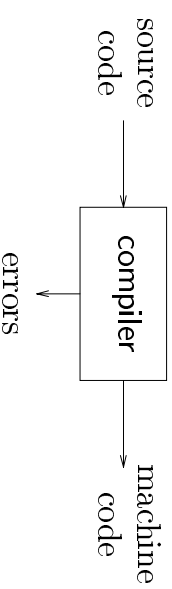
- a program that translates an *executable* program in one language into an *executable* program in another language
- the compiler typically *lowers* the level of abstraction of the program
- for “optimizing” compilers, we also expect the program produced to be *better*, in some way, than the original

Compilers are large, complex pieces of software. By working on compilers, you’ll learn to use

- programming tools (compilers, debuggers)
- program-generation tools (JLex, CUP)
- software libraries (Java class libraries)

Hopefully you will also improve your programming and software engineering skills.

Abstract view

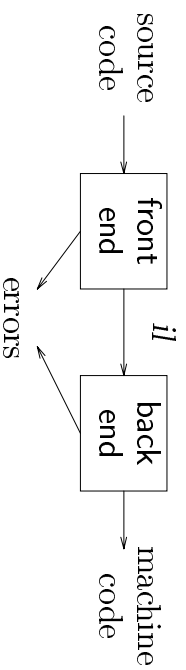


Implications:

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

Big step up from assembler – higher level notations

Traditional two pass compiler



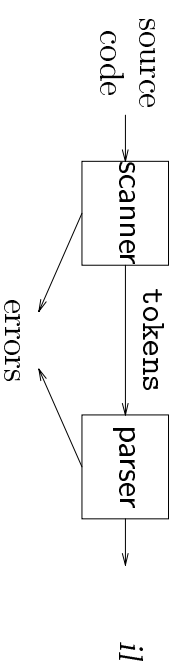
Implications:

- intermediate language (*il*)
- front end maps legal code into *il*
- back end maps *il* onto target machine
- simplify retargeting
- allows multiple front ends
- multiple passes \Rightarrow better code

Front end is $O(n)$ or $O(n \log n)$

Back end is NP-Complete

Front end

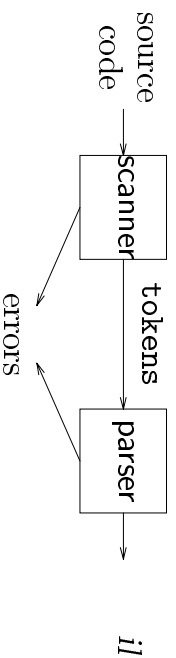


Responsibilities:

- recognize legal procedure
- report errors
- produce *il*
- preliminary storage map
- shape the code for the back end

Much of front end construction can be automated

Scanner



Scanner

- maps characters into *tokens* – the basic unit of syntax
 - $x = x + y;$
becomes
 $\langle id, x \rangle = \langle id, x \rangle + \langle id, y \rangle ;$
- character string for a *token* is a *lexeme*
- typical tokens: *number, id, +, -, *, /, do, end*
- eliminates white space (*tabs, blanks, comments*)
- a key issue is speed
 - ⇒ use specialized recognizer (*Lex*)

Specifying patterns

A scanner must recognize various parts of the language's syntax.

Some parts are easy:

white space

some combination of $\langle \backslash \rangle$ and tab

keywords and operators

specified as literal patterns — do, end

comments

opening and closing delimiters — $/ * \dots */$

Other parts are much harder:

identifiers

alphanumeric followed by k alphanumerics

($-, \$, \&, \dots$)

numbers

integers — 0 or digit from 1-9 followed by digits from 0-9

decimals — integer “.” digits from 0-9

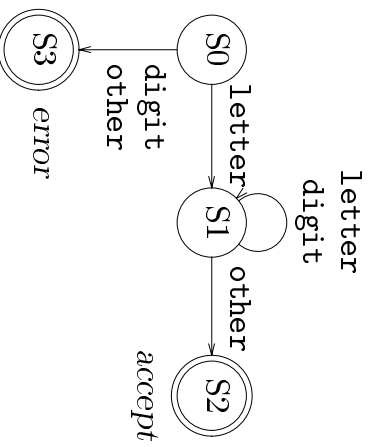
We need a powerful notation to specify these patterns.

Scanner Construction

Patterns can be specified using *regular expressions* (RE)

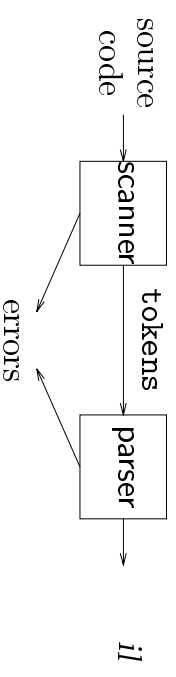
- *letter* $\rightarrow (a | b | c | \dots | z | A | B | C | \dots | Z)$
- *digit* $\rightarrow (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)$
- *identifier* $\rightarrow letter (letter | digit)^*$

Regular expressions can then be used to construct *deterministic finite automaton* (DFA)



Scanners are constructed by translating REs to DFAs, then implementing the DFAs

Parser



Parser:

- recognize context-free syntax
- guide context-sensitive analysis
- construct *il*(s)
- produce meaningful error messages
- attempt error correction

Parser generators mechanize much of the work

Grammar

Context-free syntax is specified with a *grammar*.

$$\begin{aligned} \langle \text{sheep noise} \rangle & ::= \text{baa} \\ & \quad | \text{baa } \langle \text{sheep noise} \rangle \end{aligned}$$

This grammar defines the set of noises that a sheep makes under normal circumstances.

The format is called *Backus-Naur form*. (BNF)

Formally, a grammar $G = (S, N, T, P)$

S is the *start symbol*

N is a set of *non-terminal symbols*

T is a set of *terminal symbols*

P is a set of *productions* or *rewrite rules*

$$(P : N \rightarrow N \cup T)$$

Substitution

Context free syntax can be put to better use.

1		$\langle \text{goal} \rangle$	$::=$	$\langle \text{expr} \rangle$
2		$\langle \text{expr} \rangle$	$::=$	$\langle \text{expr} \rangle \langle \text{op} \rangle \langle \text{term} \rangle$
3				$\langle \text{term} \rangle$
4		$\langle \text{term} \rangle$	$::=$	number
5				id
6		$\langle \text{op} \rangle$	$::=$	$+$
7				$-$

This grammar defines simple expressions with addition and subtraction over the tokens id and number .

$$S = \langle \text{goal} \rangle$$
$$T = \text{number, id, +, -}$$
$$N = \langle \text{goal} \rangle, \langle \text{expr} \rangle, \langle \text{term} \rangle, \langle \text{op} \rangle$$
$$P = 1, 2, 3, 4, 5, 6, 7$$

Derivations

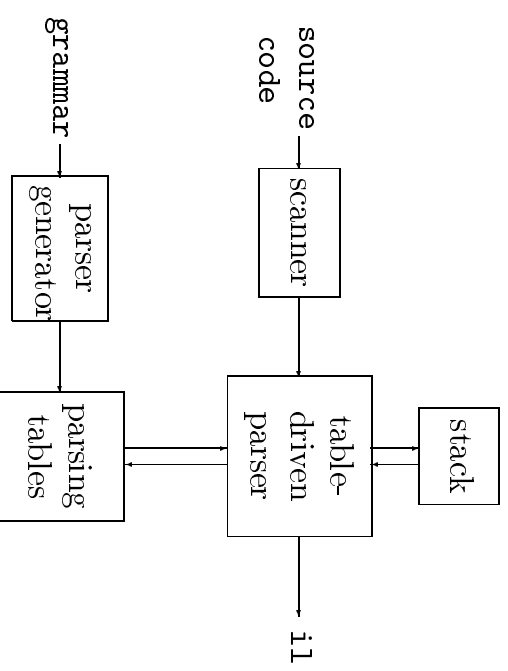
Given a grammar, valid sentences can be derived by repeated substitution.

Prod'n.	Result
1	$\langle \text{goal} \rangle$
2	$\langle \text{expr} \rangle$
5	$\langle \text{expr} \rangle \langle \text{op} \rangle \langle \text{term} \rangle$
7	$\langle \text{expr} \rangle - y$
2	$\langle \text{expr} \rangle \langle \text{op} \rangle \langle \text{term} \rangle - y$
4	$\langle \text{expr} \rangle \langle \text{op} \rangle 2 - y$
6	$\langle \text{expr} \rangle + 2 - y$
3	$\langle \text{term} \rangle + 2 - y$
5	$x + 2 - y$

To recognize a valid sentence in some *cfg*, we reverse this process and build up a *parse*.

Parser Construction

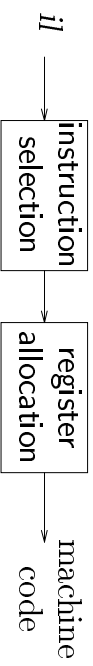
For many grammars, we can generate table-driven parsers which use a DFA and *stack*



Parsers can also perform *actions* during each reduction to

- collect information for type checking
- generate intermediate code

Back end

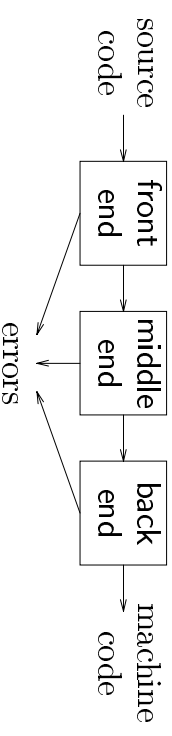


Responsibilities

- translate *il* into target machine code
- choose instructions for each *il* operation
- decide what to keep in registers at each point
- ensure conformance with system interfaces

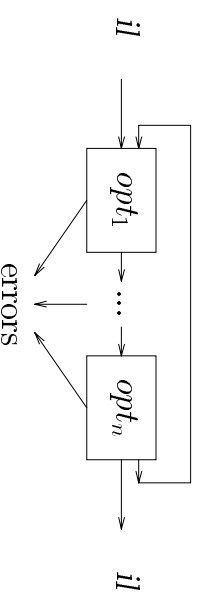
Automation has been less successful here

Optimizing compilers



Code Improvement

- analyzes and changes *il*
- goal is to reduce runtime
- must preserve values



Modern optimizers are usually built as a set of passes.