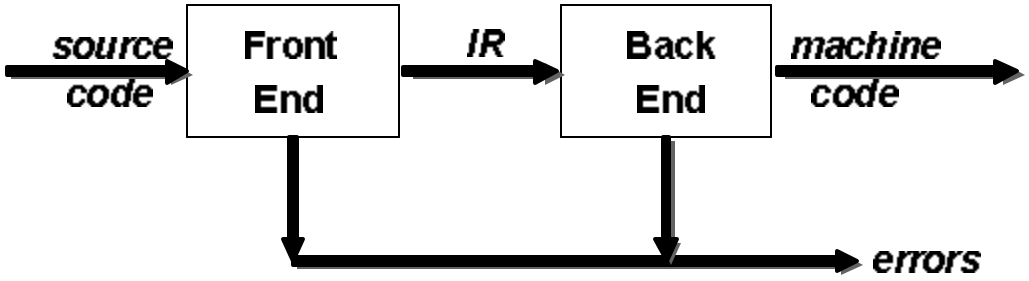
Lecture 2

Two-pass Compiler

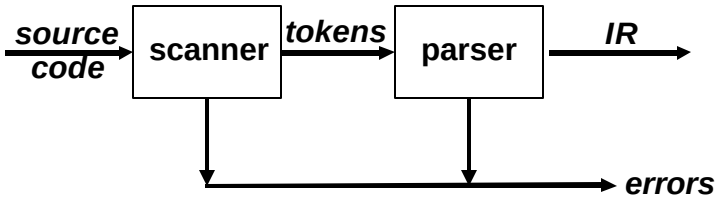


The figure above shows the structure of a two-pass compiler. The front end maps legal source code into an intermediate representation (IR). The back end maps IR into target machine code. An immediate advantage of this scheme is that it admits multiple front ends and multiple passes.

The algorithms employed in the front end have polynomial time complexity while majority of those in the backend are NP-complete. This makes compiler writing a challenging task.

Let us look at the details of the front and back ends.

Front End



The front end recognizes legal and illegal programs presented to it. When it encounters errors, it attempts to report errors in a useful way. For legal programs, front end produces IR and preliminary storage map for the various data structures declared in the program. The front end consists of two modules:

1. Scanner
2. Parser

Scanner

The scanner takes a program as input and maps the character stream into "words" that are the basic unit of syntax. It produces pairs – a word and its part of speech. For example, the input

x = x + y

becomes

<id,**x**>
<assign,**=**>
<id,**x**>
<op,**+**>
<id,**y**>

We call the pair "<token type, word>" a *token*. Typical tokens are: *number*, *identifier*, +, -, *new*, *while*, *if*.

Parser

The parser takes in the stream of tokens, recognizes context-free syntax and reports errors. It guides context-sensitive ("semantic") analysis for tasks like type checking. The parser builds IR for source program.

The syntax of most programming languages is specified using Context-Free Grammars (CFG). Context-free syntax is specified with a grammar $G=(S,N,T,P)$ where

- S is the *start* symbol
- N is a set of *non-terminal* symbols
- T is set of *terminal* symbols or words
- P is a set of *productions* or rewrite rules

For example, the Context-Free Grammar for arithmetic expressions is

1.	*goal*	?	*expr*
2.	*expr*	?	*expr op term*
3.		|	*term*
4.	*term*	?	number
5.		|	id
6.	*op*	?	+
7.		|	–

For this CFG,

S = goal

T = { number, id, +, -}
N = { *goal*, *expr*, *term*, *op*}
P = { 1, 2, 3, 4, 5, 6, 7}

Given a CFG, we can *derive* sentences by repeated substitution. Consider the sentence

x + 2 – y

Production	***Result***
	goal
1: *goal* ? *expr*	*expr*
2: *expr* ? *expr op term*	*expr op term*
5: *term* ? id	*expr op* y
7: *op* ? –	*expr* – y
2: *expr* ? *expr op term*	*expr op term* – y
4: *term* ? number	*expr op* 2 – y
6: *op* ? +	*expr* + 2 – y
3: *expr* ? *term*	*term* + 2 – y
5: *term* ? id	x + 2 – y

To recognize a valid sentence in some CFG, we *reverse* this process and build up a *parse*, thus the name “parser”.