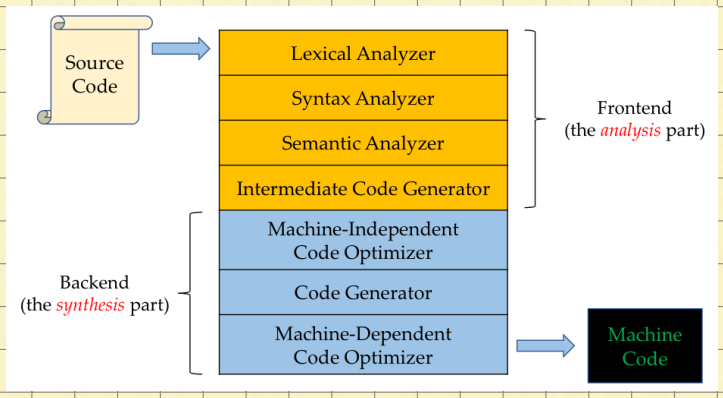
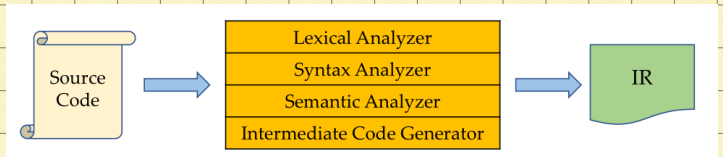
The Structure of a Compiler



Fronted



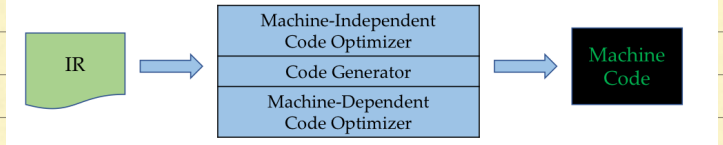
- Breaks up the source program into constituent pieces and imposes a grammatical structure on them
- Uses the grammatical structure to create an intermediate representation (IR) of the source program
- Collect the information about the source program and stores it in a data structure called *symbol table* (will be passed to backend with IR)

将源码分成连续片断并增加语法结构

由语法结构创建中间表示

创建符号表

Backend



- Constructs the target program (typically, in machine language) from the IR and the information in the symbol table
- Performs code optimizations during the process*

* Lexing and parsing are most complex and expensive in the early days, while in today, optimization dominates all other phases and lexing and parsing are very cheap.

Lexical Analysis (Scanning, 词法分析)

- The lexical analyzer (lexer/tokenizer/scanner) breaks down the source code into a sequence of "lexemes" (词素) or "words"
- For each lexeme, produce a "token" (词法单元) in the form:

```
<token-name, attribute-value>
```

An abstract symbol that is used during syntax analysis

Points to an entry in the symbol table. Info in the table entry is for semantic analysis and code generation.

将源码分割成一系列的词素.

Lexemes vs. Tokens

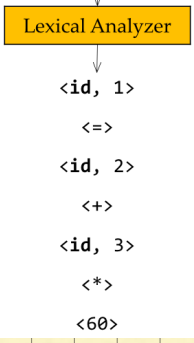
- A **lexeme** is a string of characters that is a lowest-level syntactic unit in the programming language
  - "words" and punctuation of the programming language (**instance**)
- A **token** is a syntactic category representing a class of lexemes
  - **In English:** Noun, Verb, Adjective…
  - **In programming language**: Identifier, Keyword, Whitespace… (**pattern**)

最小语法单元

词素的语法分类.

Example

```
position = initial + rate * 60
```

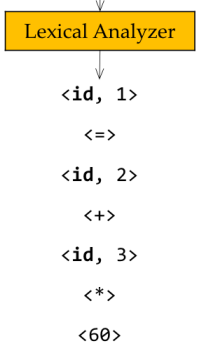


SYMBOL TABLE

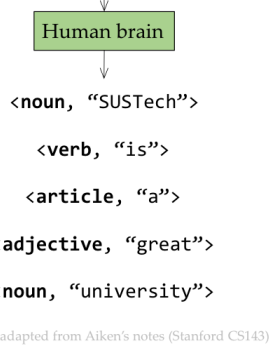
1	position	...
2	initial	...
3	rate	...

Note: <=>, <+>, <*>, <60> are not in the defined form. This is for notational convenience. <=> could have been <assign, -> and <60> could have been <number, 4>.

```
position = initial + rate * 60
```




```
SUSTech is a great university
```



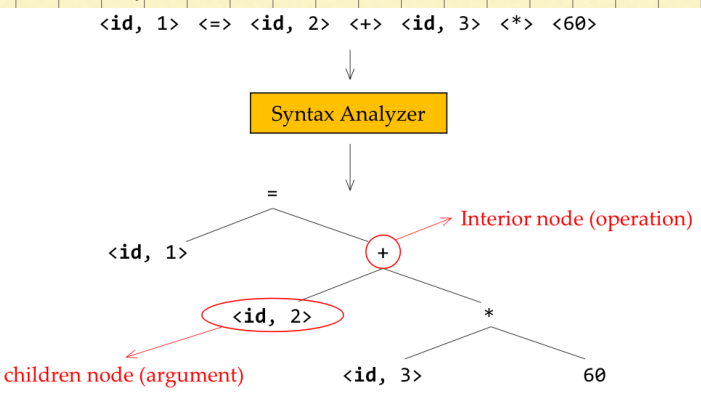
Example adapted from Aiken's notes (Stanford CS143)

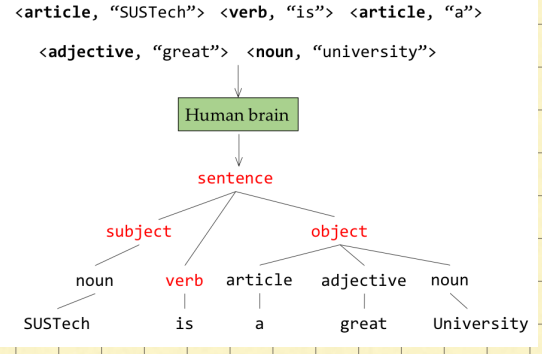
Syntax Analysis (Parsing, 语法分析)

根据token信息构建语法树.

- The syntax analyzer (parser) uses the token names produced by the lexer to create an intermediate representation that depicts the grammar structure of the token stream, typically a *syntax tree*
- Each interior node represents an operation and the children of the node represent the arguments of the operation

Example





Semantic Analysis (语义分析)

Syntax tree → Semantic Analyzer → Syntax tree

- The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition
- Also gathers type information for type checking, type conversion, and intermediate code generation

Semantics

- The syntax of a programming language describes the proper form of its programs

语法规定了程序的正常形式

- The semantics of a programming language describes the meaning of its programs, i.e., what each program does when it executes

语义规定了程序的意思

Semantic Analysis in Programming

- Understanding the meaning of a program is very hard ☹
- Compilers perform only very limited analysis to catch semantic inconsistencies.

```
1. {
2.   int Jack = 3;
3.   {
4.     int Jack = 4;
5.     print Jack;
6.   }
7. }
```

Which value will be printed?

Programming languages define strict rules to avoid ambiguities.

Compiler will bind Jack at line 5 to its inner definition at line 4.

Type Checking

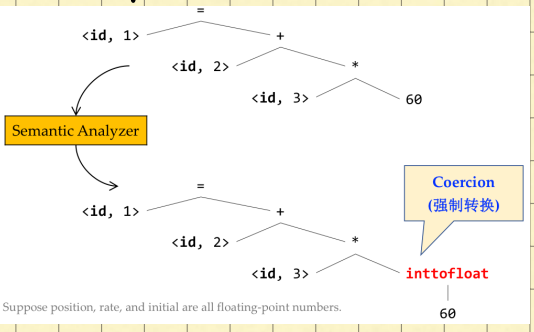
- An important part of semantic analysis is type checking
- Compilers check that each operator has matching operands (of correct types)

Example: Many language definitions require an array index to be an integer.

```
double x = 3.2;
int[] nums = new int[5];
nums[x] = 6;
```

Compilers should report an error!

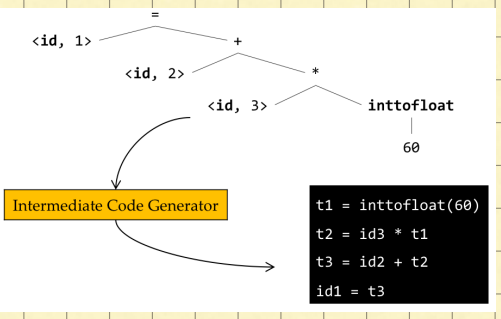
Example



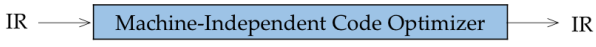
Intermediate Code Generation

- After semantic analysis, compilers generate an intermediate representation, typically *three-address code (三地址码)*
  - Assembly-like instructions with three operands per instruction
  - Each operand acts like a register
  - Each assignment instruction has at most one operator on the RHS
  - Easy to translate into machine instructions of the target machine

Example



Machine-Independent Code Optimization



- Akin to article editing/revising in English
- Improve the intermediate code for better target code
  - Run faster
  - Use less memory
  - Shorter code
  - Consume less power …

Example

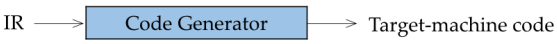
```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

1. 60 is a constant integer value. Its conversion to floating-point can be done once and for all at compile time
2. t2 and t3 are only used for value transmitting

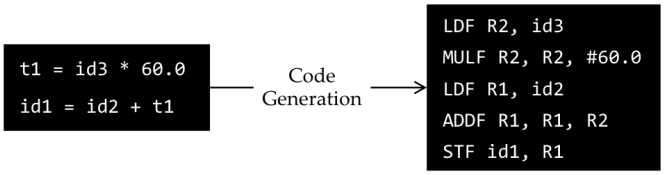
Optimization

```
t1 = id3 * 60.0
id1 = id2 + t1
```

Code Generation



- Map IR to target language, analogous to human translation
- It is crucial to allocate register and memory to hold values

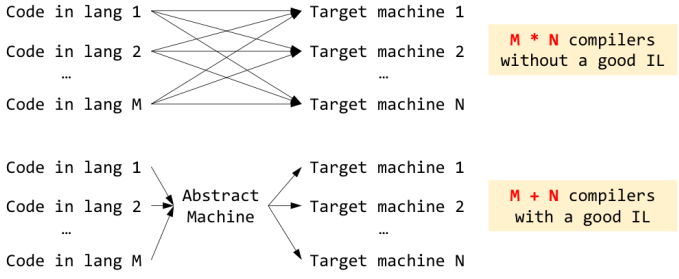


Symbol Table Management

- Performed by the frontend, symbol table is passed along with the intermediate code to the backend
- Record the variable names and various attributes
  - storage allocated, type, scope
- Record the procedure names and various attributes
  - the number and type of arguments
  - the way of passing arguments (by value or by reference)
  - the return type

Intermediate Language (IL)

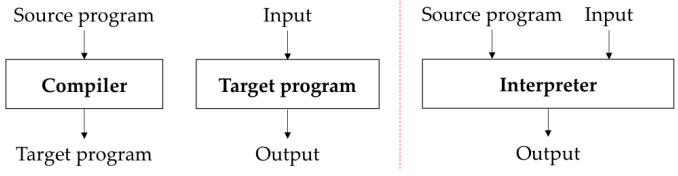
- Intermediate code is in IL (e.g., three-address code)
- A good IL eases compiler implementation



減少compiler數量

Compilers vs. Interpreters

1 A complier translates source programs written in high-level languages into machine codes that can run directly on the target computer.
An interpreter directly executes each statement in the source code, without requiring the program to have been compiled into machine codes.



2 Interpreters often take less time to analyze the source code: they simply parse each statement and execute it (e.g., Python code).

In comparison, compilers typically analyze the relationships among statements (e.g., control and data flows) to enable optimizations.

3 Interpreters continue executing a program until the first error is met, in which case they stop.

For compiled languages, programs are executable only after they are successfully compiled.