CISC 3160
Programming Languages and Compilers

Topics

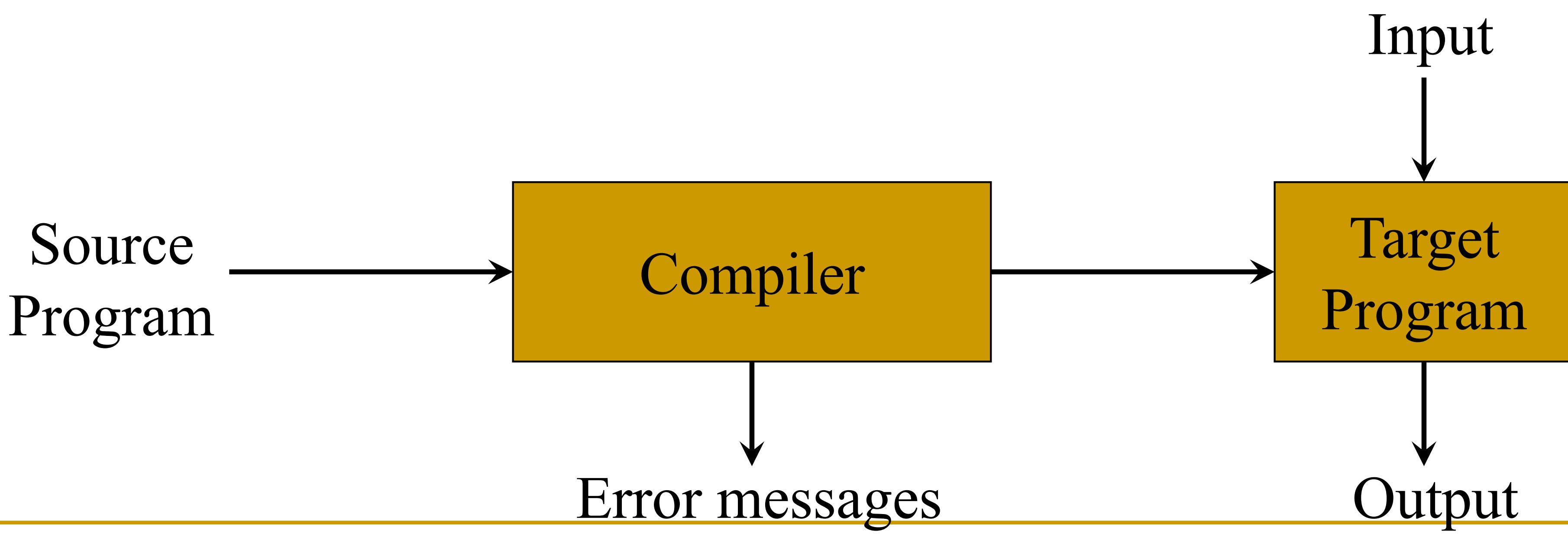
- Compilers
  - Rregular expressions and context-free grammars
  - Scanning and parsing
  - run-time systems and memory management
- Programming paradigms
  - Imperative programming, object-oriented programming, functional programming, logic and constraint programming, scripting languages, concurrent programming
- Programming language examples
  - Java, C/C++, Python, Haskell, and Picat

Resources

- Compilers: Principles, Techniques, and Tools (2nd Edition), by Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman.
- Data Structures with C++ Using STL, 2nd ed., by - William H. Ford and William R. Topp, Prentice-Hall.
- Java Online Tutorials
- OOP Wiki
- Introduction to Python, by Guido van Rossum
- A Gentle Introduction to Haskell, by Paul Hudak, John Peterson, and Joseph Fasel.
- Constraint Solving and Planning with Picat, by Neng-Fa Zhou, Hakan Kjellerstrand, and Jonathan Fruhman.

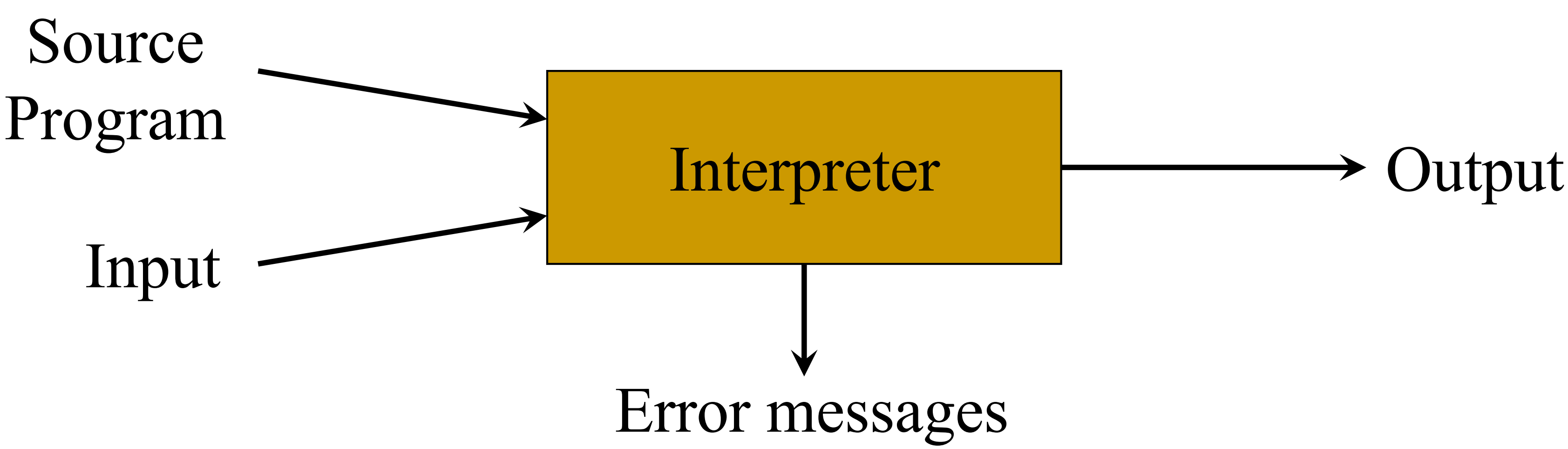
Compilers

- "*Compilation*"
  - Translation of a program written in a source language into a semantically equivalent program written in a target language

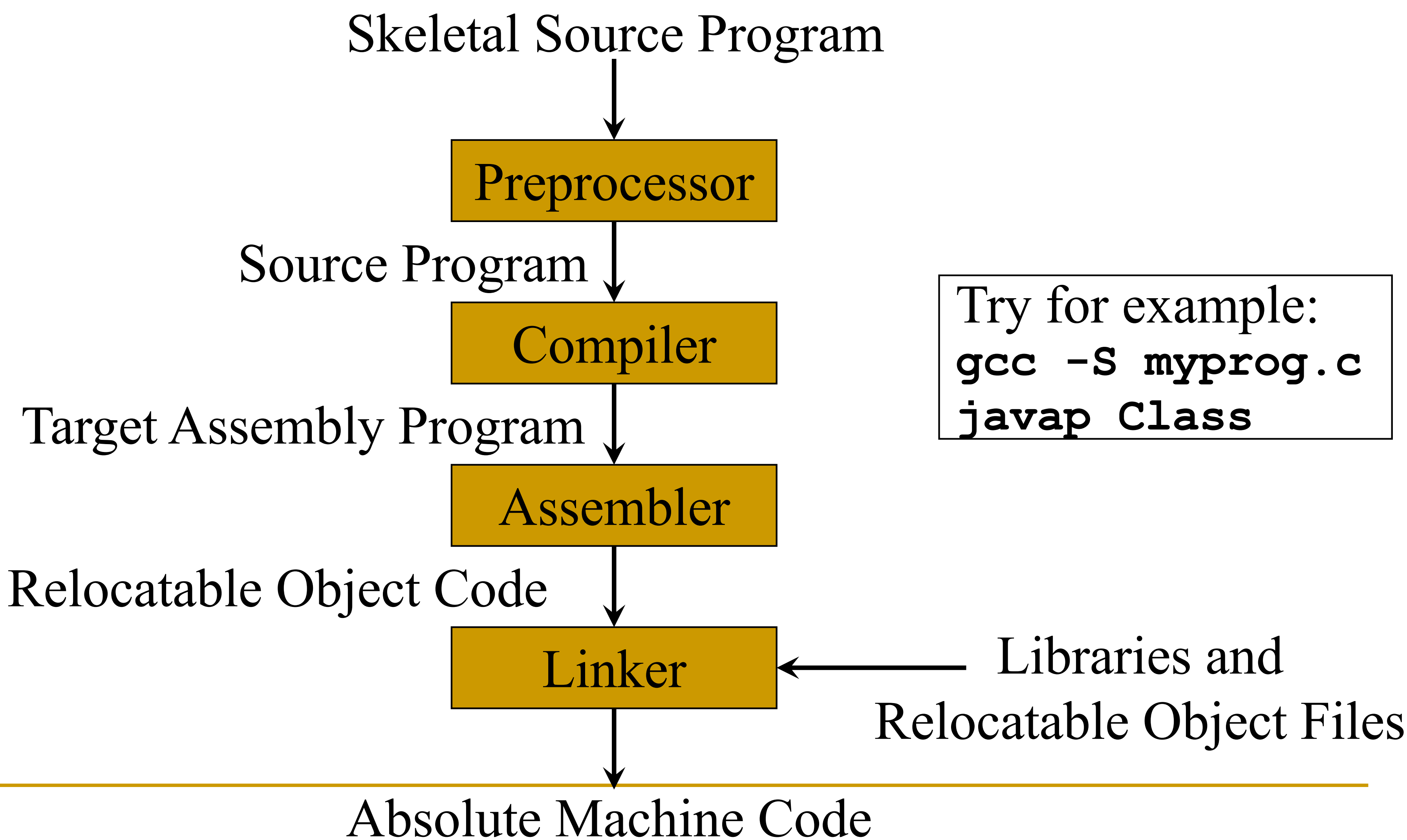


Interpreters

- "*Interpretation*"
  - Performing the operations implied by the source program



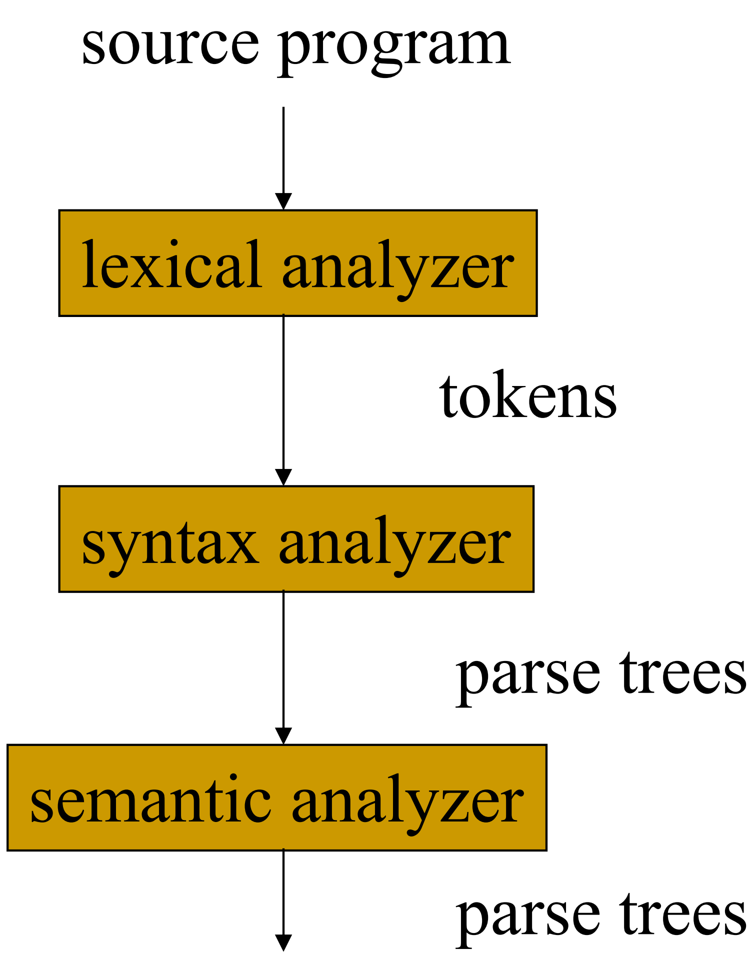
Preprocessors, Compilers, Assemblers, and Linkers



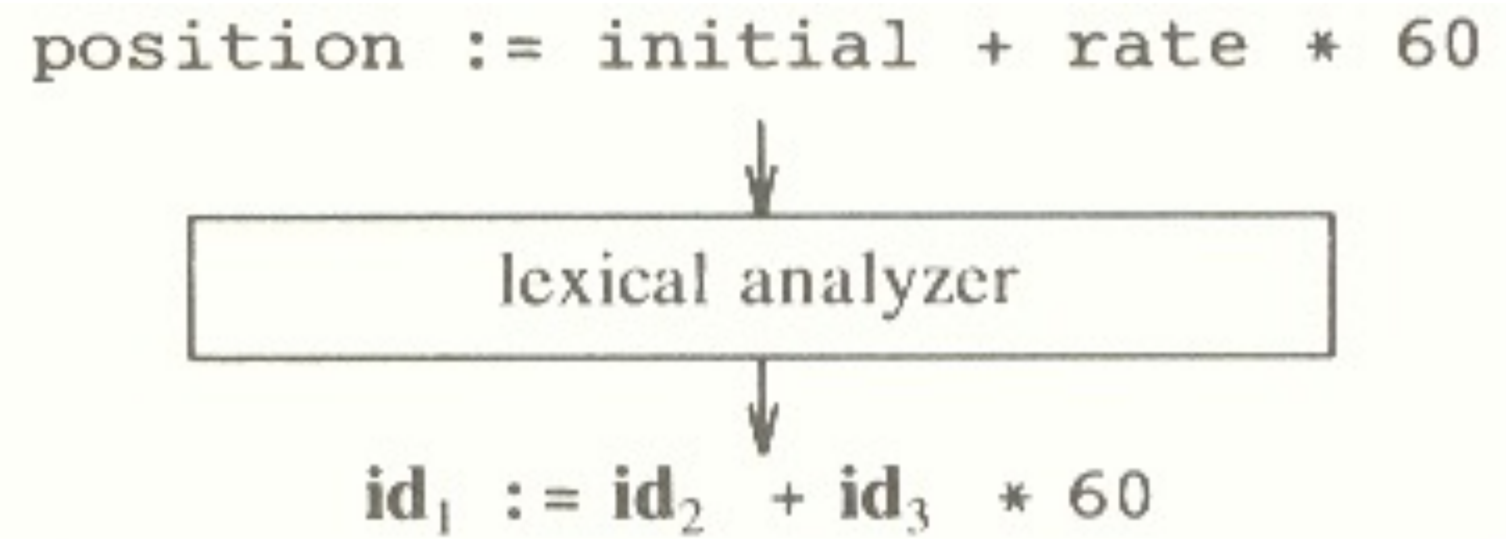
Try for example:

```
gcc -S myprog.c
javap Class
```

Analysis of Source Programs

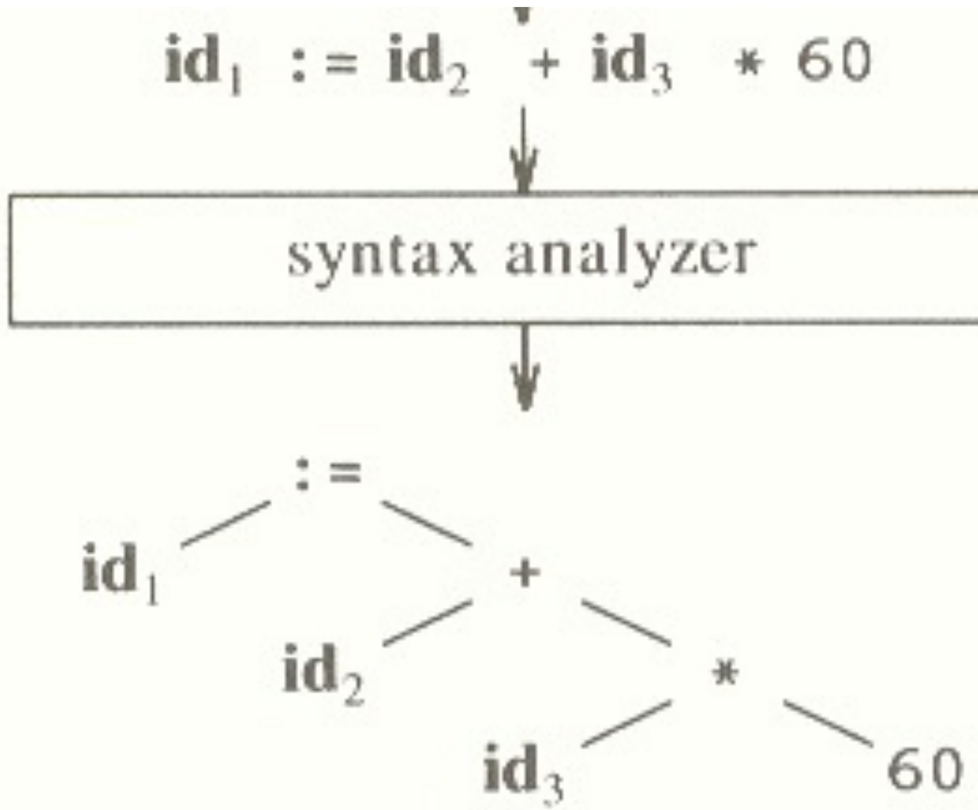


Lexical Analysis



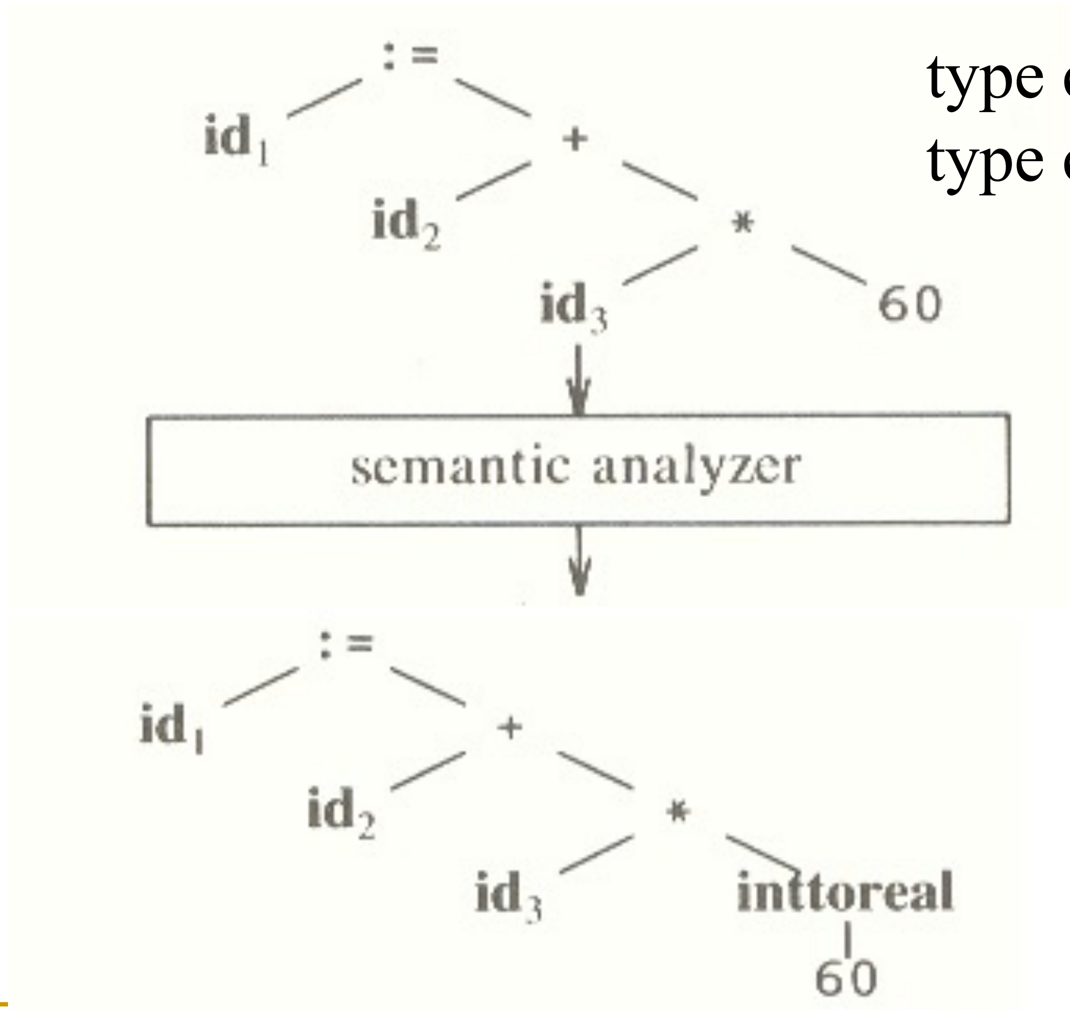
tokens

Syntax Analysis



parse tree

Semantic Analysis



type checking
type conversion

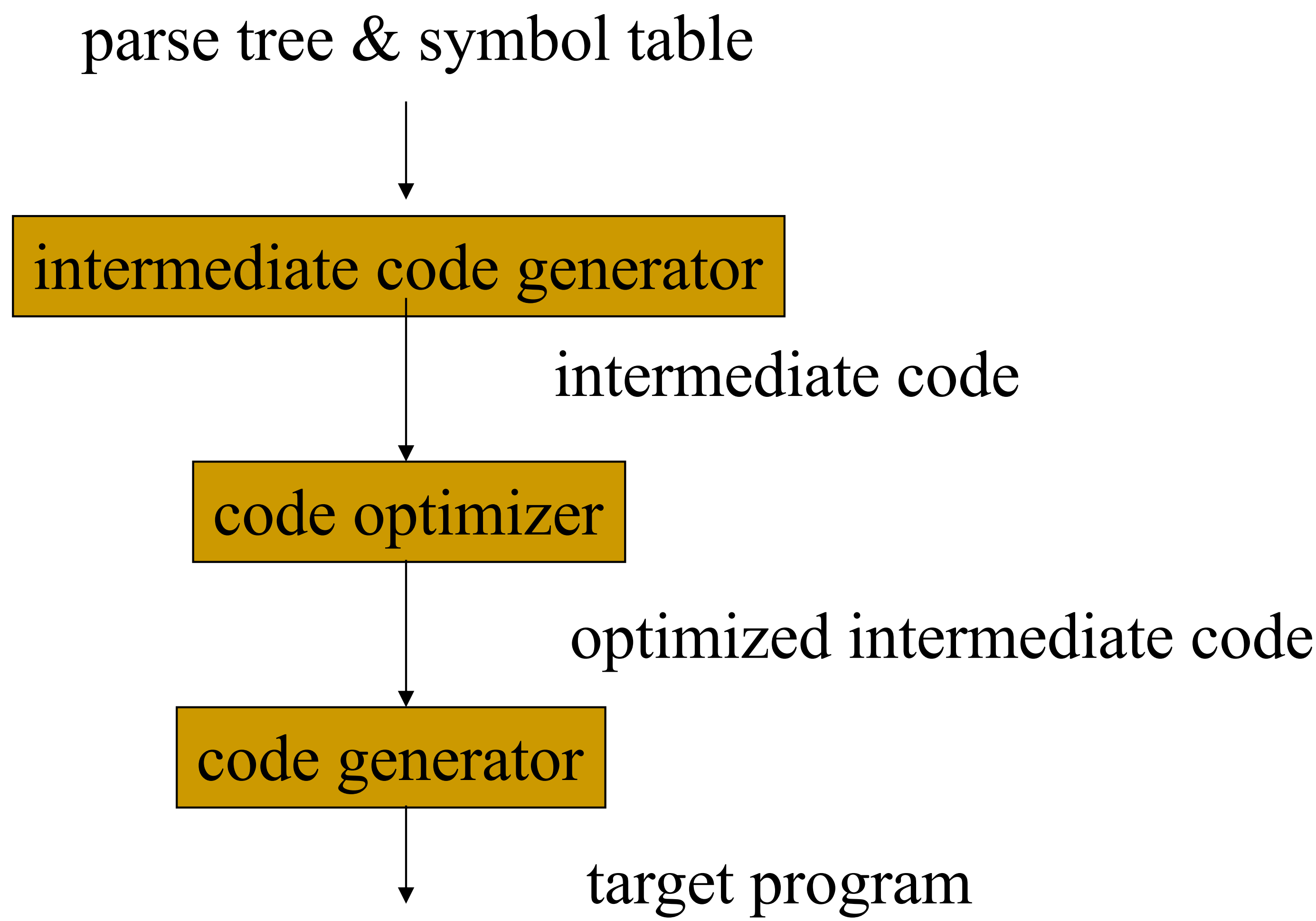
Symbol Table

SYMBOL TABLE

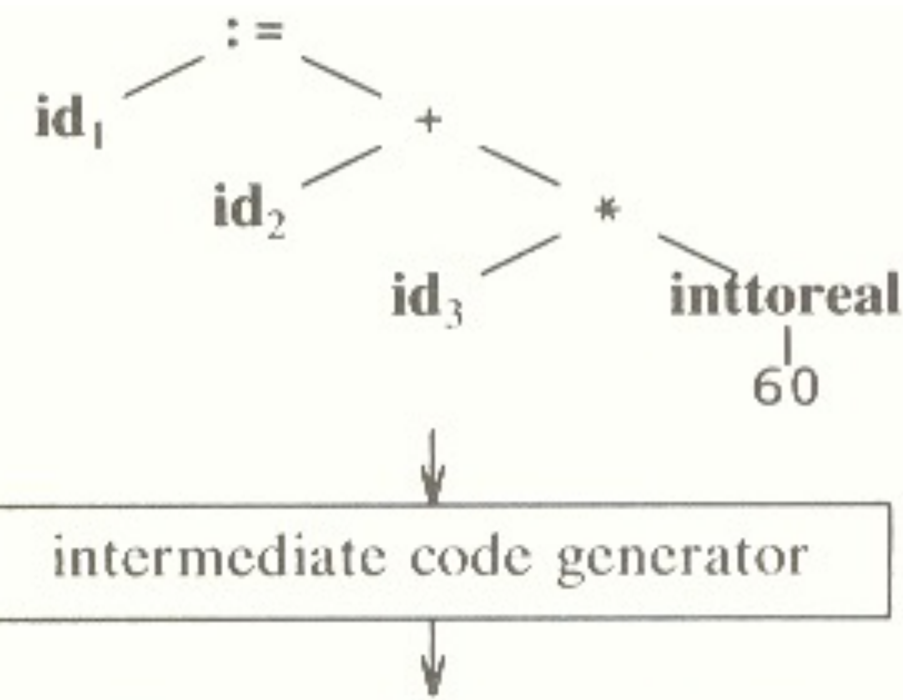
1	position	. . .
2	initial	. . .
3	rate	. . .
4		

- There is a record for each identifier
- The attributes include name, type, location, etc.

Synthesis of Object Code



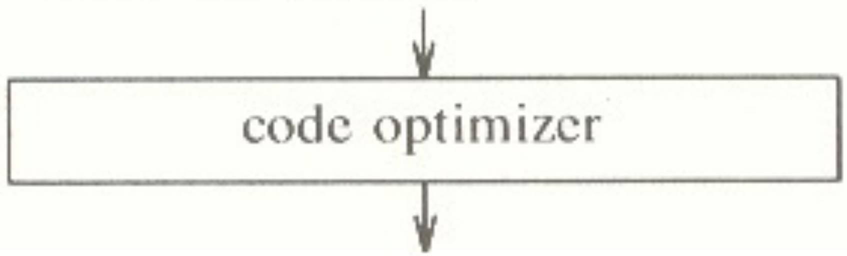
Intermediate Code Generation




```
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```

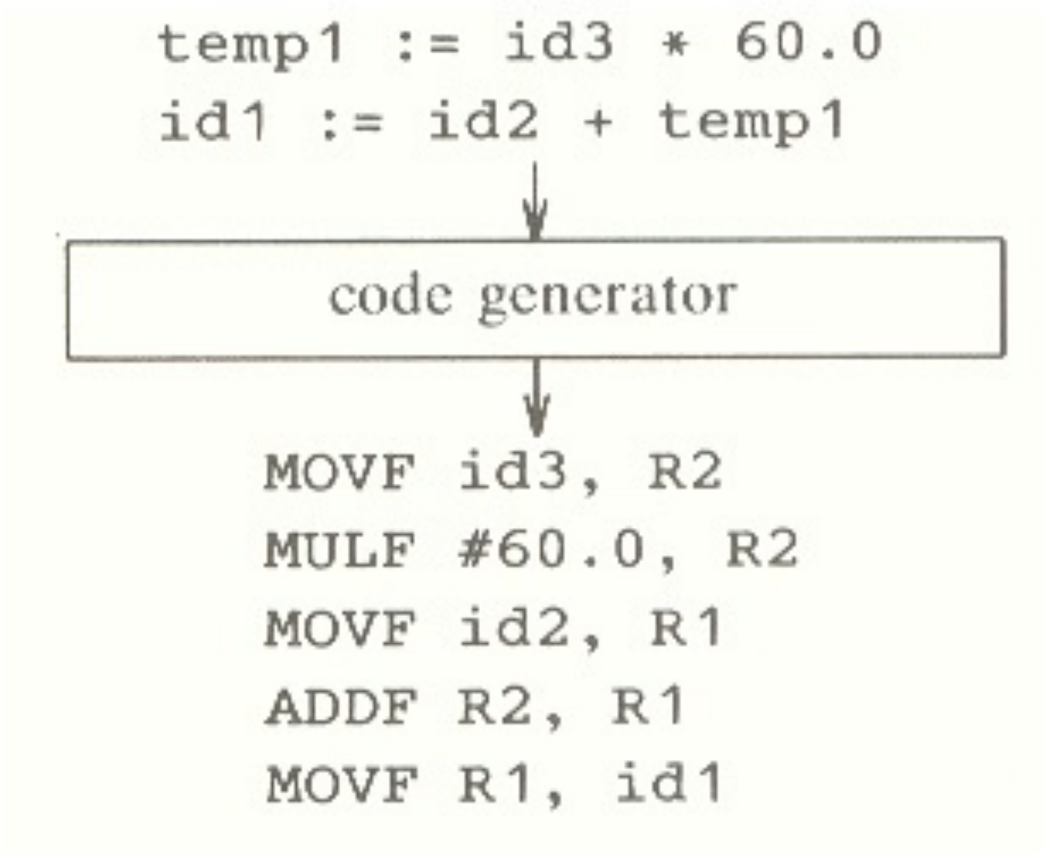
Code Optimization

```
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```




```
temp1 := id3 * 60.0
id1 := id2 + temp1
```

Code Generation



Qualities of a Good Compiler

What qualities would you want in a compiler?

- generates correct code (first and foremost!)
- generates fast code
- conforms to the specifications of the input language
- copes with essentially arbitrary input size, variables, etc.
- compilation time (linearly)proportional to size of source
- good diagnostics
- consistent optimisations
- works well with the debugger

Principles of Compilation

The compiler must:

- *preserve the meaning of the program being compiled*.
- *"improve" the source code in some way*.

Other issues (depending on the setting):

- Speed (of compiled code)
- Space (size of compiled code)
- Feedback (information provided to the user)
- Debugging (transformations obscure the relationship source code vs target)
- Compilation time efficiency (fast or slow compiler?)

Why study Compilation Technology?

- Success stories (one of the earliest branches in CS)
  - Applying theory to practice (scanning, parsing, static analysis)
  - Many practical applications have embedded languages (eg, tags)
- Practical algorithmic & engineering issues:
  - Approximating really hard (and interesting!) problems
  - Emphasis on efficiency and scalability
  - Small issues can be important!
- Ideas from different parts of computer science are involved:
  - AI: Heuristic search techniques; greedy algorithms - Algorithms: graph algorithms - Theory: pattern matching - Also: Systems, Architecture
- Compiler construction can be challenging and fun:
  - new architectures always create new challenges; success requires mastery of complex interactions; results are useful; opportunity to achieve performance.

Uses of Compiler Technology

- Most common use: translate a high-level program to object code
  - Program Translation: binary translation, hardware synthesis, …
- Optimizations for computer architectures:
  - Improve program performance, take into account hardware parallelism, etc…
- Automatic parallelisation or vectorisation
- Performance instrumentation: e.g., -pg option of cc or gcc
- Interpreters: e.g., Python, Ruby, Perl, Matlab, sh, …
- Software productivity tools
  - Debugging aids: e.g, purify
- Security: Java VM uses compiler analysis to prove "safety" of Java code.
- Text formatters, just-in-time compilation for Java, power management, global distributed computing, …

Key: Ability to extract properties of a source program (analysis) and transform it to construct a target program (synthesis)

Exercises

1.1.1 Exercises for Section 1.1

Exercise 1.1.1: What is the difference between a compiler and an interpreter?

Exercise 1.1.2: What are the advantages of (a) a compiler over an interpreter (b) an interpreter over a compiler?

Exercise 1.1.3 : What advantages are there to a language-processing system in which the compiler produces assembly language rather than machine language?

Exercise 1.1.4: A compiler that translates a high-level language into another high-level language is called a *source-to-source* translator. What advantages are there to using C as a target language for a compiler?

Exercise 1.1.5: Describe some of the tasks that an assembler needs to perform.