Compiler Design

Introduction

Translators

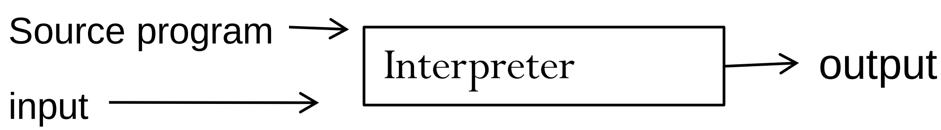
- Translators:
  - It is a **program**
  - It takes *input* as a program written in one programming language (*source language*)
  - produces *output* as a program in another language (*the object or target language*)
- Types of translators
  - Compilers:
    - *It is a translator*
    - *The input is a source language* -> is a high level language
    - *The output is object language* -> is an assembly language or machine language

Translators cont…

- *source program* → Compiler → *target program*
- *If the target program is a machine language program then it can be called by the user to process inputs to outputs*
- *input* → Target program → *output*
- Assembler
  - *It is a translator*
  - *source language* -> is an assembly language
  - *target language* -> machine language

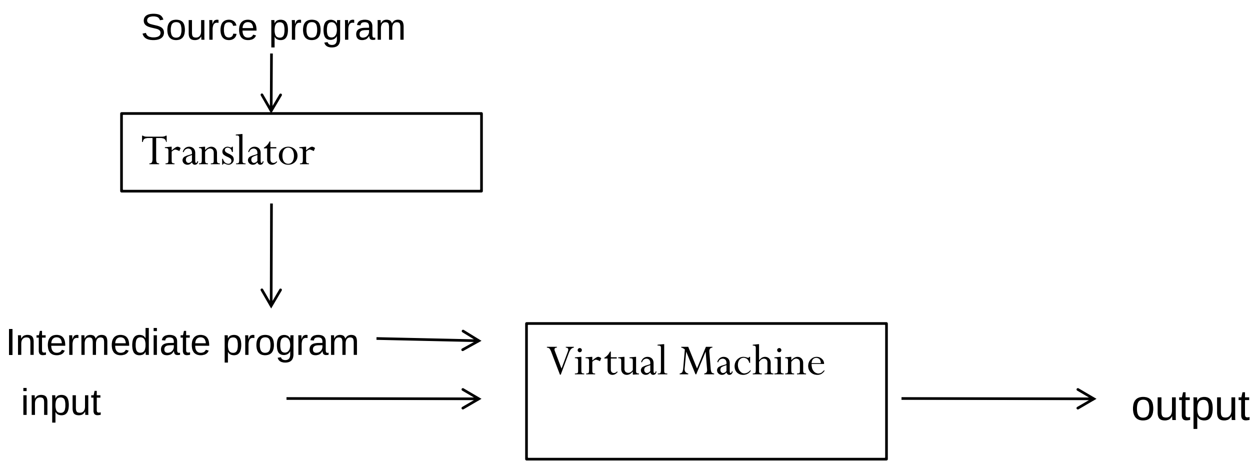
Translators cont…

- Interpreters
  - Instead of producing a target program as a translation, it directly execute the operations specified in the source program on inputs supplied by the user.



- Compiler is usually faster than an interpreter at mapping inputs to outputs
- An interpreter however can give better diagnostics than a compiler
  - since it executes the source program statement by statement

Translators cont…



A hybrid Compiler

- Java language processors combine both compilation and interpretation.
- A java source program is first compiled into an intermediate form called *bytecodes*.
- The *bytcodes* is then interpreted by a virtual machine

Phases of a compiler

- The compilation process is so complex that we cannot do it in one step
- Therefore it is necessary to *partition* the compilation process into a series of *sub-processes* called *phases*

Phases of a compiler cont…

- What is actually a phase?
  - it is a logically cohesive operation
  - that takes as *input* one *representation* of the source program
  - and produces as *output* another *representation*

First phase

- called the *lexical analyzer* or *scanner*
  - from the source program
  - it separates out the characters into **groups (atomic units)** that logically belong together
  - these *groups* are called **tokens**
  - Identifiers, keywords, constants, punctuation, labels, operators etc
  - what is called a token depends on
    - the language/ compiler designer

First phase cont…

- two kinds of tokens
  - *specific strings* such as **IF** or **;**
  - *classes of strings* such as identifiers, constants or labels
  - E.g token **;** has a type but no value
  - token MAX has a type “identifier” and value MAX

First phase cont…

- token **identifiers**
  - num, dob, x etc
- token **operator**
  - <, <=, + , (
- the output of this phase is a **stream of tokens**
- the tokens in this stream can be represented by **codes** which may be regarded as integers

First phase cont…

- Example
  - DO by 1
  - + by 2
  - “identifier” by 3 -> in this case a second quantity telling which identifier is used is passed along with the code

First phase cont…

- the lexical analyzer and the following phase are often grouped into the same pass
  - the lexical analyzer behaves as a co-routine or under the control of the parser
  - lexical analyzer returns the **code** of the token (and the **value** if the token is like an identifier etc)

First phase cont…

- The lexical analyzer calls a **bookkeeping** routine which *installs* the actual value if it is not already there
- Then it passes the two components of the token to the parser
  - the first is the token type (identifier)
  - and the second is the value (a pointer to the place in the symbol table reserved for the specific value)

First phase cont…

- Finding Tokens
  - the lexical analyzer may be required to search many characters beyond the next token in order to actually determine what the next token actually is
    - e.g IF (5.EQ.MAX) GOTO 100
    - IF(5.EQ.MAX)GOTO100
    - **if ([const**,341] **eq** [id,729] **) GOTO [label**,554]

First phase cont…

341	constant, integer, value = 5
554	label, value = 100
729	variable, integer, value = MAX

Symbol Table

Second phase

- called the *syntax analyzer*
- it has two functions

1. it checks the syntactic correctness of the input program (i.e the tokens appearing in its input occur in patterns specified by the source language)
   - e.g A + / B translates to **id + / id**
2. it also imposes on the tokens a tree-like structure that is used by subsequent phases of the compiler
   - e.g A / B * C (has **two** interpretations)
     1. Divide A by B then multiply by C
     2. Multiply B by C and then use the result to divide A

Second phase cont…

- Each of the two interpretations can be represented by a **parse tree** (a diagram which exhibits the syntactic structure of the expression)
- The language specification must tell us which interpretation is to be used
- These rules form the syntactic specification of a programming language

Second phase cont…

- Example
  - groups the **tokens** together into **syntactic structures**
  - E.g 3 tokens A+B (i.e A operator + and B) might be grouped into a syntactic structure called an *expression*
  - The syntactic structure is a tree whose leaves are the tokens
  - The interior nodes of the tree represent strings of tokens that logically belong together

Third phase

- called the *intermediate code generator*
- Transforms the parse tree into an intermediate-language representation of the source program
- it uses the *tree* to create a stream of simple instructions (not assembly code)
  - e.g- one type of intermediate language is called 'Three-Address code"
  - uses instructions with one operator and a small number of operands
  - A:=B **op** C
  - E.g A / B * C

Third phase cont…

- It also needs unconditional and simple conditional branching statements
- i.e at most one relation is tested to determine whether or not a branch is to be made
- Higher level flow of control while-do/if-then-else statements are translated into these lower-level conditional three-address statements
- E.g.,

```
while A>B & A<=2*B-5 do
      A:=A + B
```

Third phase cont…

- Advantage:
  - helps the compiler for generating code from one processor to another
  - this language is supported by most processors
  - then code generators can use this language to generate target code

Fourth phase

- called *code optimization*
- it is an optional phase
- designed to improve the intermediate code
  - so that the ultimate object program runs faster and/or takes less space
- its output is another intermediate code program
- Local optimization and Loop optimization

Fourth phase cont…

- Local optimization
  - Loops

  L1: If A>B goto L2

  Goto L3

  Can be replaced by

  L1: if A≤ B goto L3

  - Elimination of common sub expression
    - A:= B + C + D
    - E:= B + C + F
    - T1:=B + C
    - A:=T1 + D
    - E:=T1 + F

Fourth phase cont…

- Loop optimization
  - A typical loop improvement is to move a computation that produces the same result each iteration to a point in the program just before the loop is entered.
    - Such a computation is called loop invariant

Fifth phase

- called *code generation*
  - Converts the intermediate code into a sequence of machine instructions.
  - selecting memory locations for data
  - selecting code to access each datum
  - selecting the registers in which each computation is to be done
  - A:=B+C
  - LOAD B
  - ADD C
  - STORE A

Fifth phase cont…

- such a straightforward macro-like expansion usually produces a target program that contains many redundant **loads** and **stores**
- a code generator might keep track of the run-time contents of registers
- generate **load** and **stores** only when necessary
- a good code generator would therefore attempt to utilize these registers as efficiently as possible

Table management/Bookkeeping

- It is that part of the compiler that keeps track
  - of *names* used by the program and its information like (type i.e int or real etc)
- data structure used is called *symbol table*

Table management/Bookkeeping cont...

- A compiler needs to collect information about **all** the **data objects** that appear in the source program
  - type (int/real)
  - array size
  - how many arguments a function expect etc
- This information may be explicit (declarations)/implicit

Table management/Bookkeeping cont…

- This information is collected by the early phases of the compiler
  - lexical –enters MAX and returns pointer
  - syntax analyzer- enters integer when it encounters **integer MAX**
  - no intermediate code is generated
- Advantage
  - expression of mixed mode A+B where A is **int** and B is **float** (conversion/error)

Error Handling

- this is invoked when a flaw is detected in the source program
- It is desirable that compilation be completed on flawed programs at least through the syntax analysis phase , so that as many errors as possible can be detected in one compilation

Error Handling cont…

- The error messages should allow the programmer to determine exactly where the errors have occurred
- Errors can be encountered in all the phases of a compiler
- Once the error has been noted , the compiler must modify the input to the phase detecting the error, so that the latter can continue processing its input, looking for subsequent errors
- good error handling is difficult because
  - certain errors can mask subsequent errors
  - other errors if not properly handled, can spawn an avalanche of spurious errors

Passes

- In an implementation of a compiler
  - Activities from several phases may be grouped together into a pass that reads an input file and writes an output file.
  - E.g., front-end phases of lexical analysis, syntax analysis, semantic analysis and intermediate code generation can be grouped into one pass.
  - Code optimization might be an optional pass and there could be back-end pass consisting of code generation for a particular target machine
    - portability
  - its convenient to process the entire source program several times before generating code
  - these repetitions are called **passes**

Passes

- **Example**
  - Initial pass- output tree/intermediate code
  - next pass- process the intermediate representation by adding/altering its structure or producing a different representation
- passes may or may not correspond to phases
  - Often a pass consist of several phases.
- a compiler may be **one pass** (all phases occur during a single pass)
  - **Pascal , C**

Passes cont…

- The **number** of passes and the **grouping** of phases into passes are usually dictated by a variety of considerations to a particular language and machine
- A multi-pass compiler can be made to use less space (reusing space) but slower (because each pass reads and writes an intermediate file)