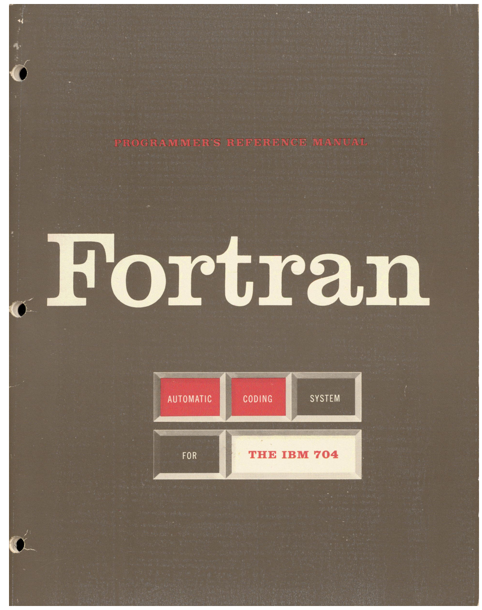
Compilers



CSE 4305 / CSE 5317
M00 Introduction
Fall 2020

M00
Introduction

First FORTRAN Manual, 1956 October 15

The IBM Mathematical Formula Translating System FORTRAN *is an automatic coding system for the IBM 704 EDPM. More precisely, it is a 704 program which accepts a* source program *written in a language — the* FORTRAN *language — closely resembling the ordinary language of mathematics, and which produces an* object program *in 704 machine language, ready to be run on a 704.*

FORTRAN *therefore in effect transforms the 704 into a machine with which communication can be made in a language more concise and more familiar than the 704 language itself. The result should be a considerable reduction in the training required to program, as well as in the time consumed in writing programs and eliminating their errors.*

Programming *with* Language

- ***Machine Code***
  - The actual ones and zeros that the processor uses to control its operation.
  - Each processor architecture has its own interpretation.
- Example is a GCD routine for the x86 architecture.

```
55 89 e5 53 83 ec 04 83 e4 f0
e8 31 00 00 00 89 c3 e8 2a 00
00 00 39 c3 74 10 8d b6 00 00
00 00 39 c3 7e 13 29 c3 39 c3
75 f6 89 1c 24 e8 6e 00 00 00
8b 5d fc c9 c3 29 d8 eb eb 90
```

Programming *with* Language

- ***Assembly Language***
  - Gives mnemonic (!) names to the instructions, registers, etc.
  - An *assembler* translates this into machine code.
  - Eventually included more than just 1-1 correspondence. (E.g., *macros*)

- Q: How was the first assembler written?

```
   pushl %ebp
   movl  %esp, %ebp
   pushl %ebx
   subl  $4, %esp
   andl  $-16, %esp
   call  getint
   movl  %eax, %ebx
   call  getint
   cmpl  %eax, %ebx
   je    C
A: cmpl  %eax, %ebx
   jle   D
   subl  %eax, %ebx
B: cmpl  %eax, %ebx
   jne   A
C: movl  %ebx, (%esp)
   call  putint
   movl  -4(%ebp), %ebx
   leave
   ret
D: subl  %ebx, %eax
   jmp   B
```

Programming *with* Language

- ***High-Level Language***
  - Gets away from any particular processor architecture. (Generally …)
  - A *compiler* translates the high-level language to (assembly which is then translated to) machine code. (Generally …)
  - At first, humans could write better assembly code than the compiler generated, so slow to catch on.
  - But, reduced the number of statements that had to be written by a factor of 20!
  - It took 18 staff-years (!) to write the first FORTRAN compiler.

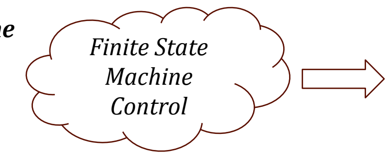
- Q: How was the first compiler written?

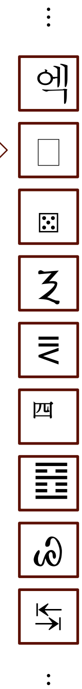
```
      FUNCTION IGCD()
      READ(5,500) IA, IB
500   FORMAT(2I5)
600   IF (IA.EQ.IB) GOTO 800
      IF (IA.GT.IB) GOTO 700
      IB = IB - IA
      GOTO 600
700   IA = IA - IB
      GOTO 600
800   IGCD = IA
      RETURN
      END(2, 2, 2, 2, 2)
```

[*Turing Complete*]

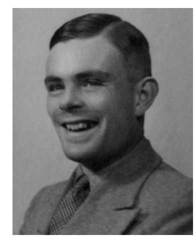
- ***Informally***, any real-world general-purpose computer or computer language can approximately simulate the *computational* aspects of any other real-world general-purpose computer or computer language.
- In other words, they are all the same; it's just that some may be more or less *convenient* for any particular *computation*.

Turing Machine





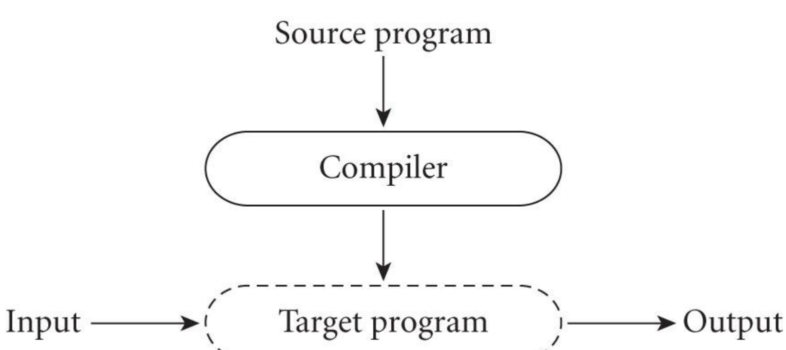
1. ***Read*** *symbol from tape.*
2. *Based on current state and symbol …*
   - [*optional*] ***Write*** *a symbol to the tape.*
   - [*optional*] ***Move*** *tape to Left or Right.*
   - ***Transition*** *to the next state.*
3. *If state is a halting state,* ***stop****.*
4. ***Goto*** *step 1.*



http://www.turingarchive.org/browse.php/K/7/9-16

Compiling a Program

- A *compiler* translates a *source* program into an equivalent *target* program and then goes away. At some later time, the target program can be executed.

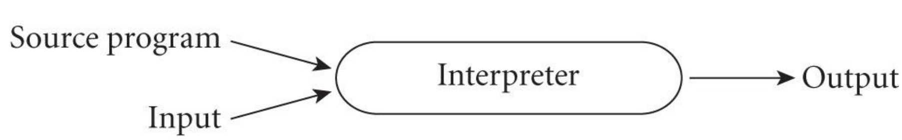


Compiling a Program

- During compilation, the compiler is the *locus of control*.
- During its own execution, the target program is the locus of control.
- Typically, the compiler is a machine language program.
- Typically, the target program is also a machine language program.
- Generally leads to best performance.
- *Translate once, run many times*.

Interpreting a Program

- An *interpreter*, on the other hand, typically stays around for the execution of the target program.
  - The interpreter is the locus of control during all parts of the execution.
  - The interpreter is in effect a *virtual machine* whose machine language is the programming language.
  - *Translate every time*.



Interpreting a Program

- Generally, interpretation is more flexible and has better diagnostics than compilation.
  - The source code of the program is still available.

- Some language features are very difficult to implement without interpretation.
  - “On-the-fly” code generation

[*Read-Eval-Print Loop (REPL)*]

- A *Read-Eval-Print Loop* is an interactive environment wherein the user types a line at a time which is then evaluated and results displayed.
  - *Read* — Get input from user.
  - *Eval* — Determine value.
  - *Print* — Display result to user.
  - *Loop* — Do again, until terminated.
- Can be created for any text-based language, Lisp, Python, Java …
- REPL is a one-liner in Lisp for Lisp:

  `(loop (print (eval (read))))`
- Lot of REPLs available *live* and *on-line*:

  `http://joel.franusic.com/Online-REPs-and-REPLs/`

```
(base) dalioba@achpiel:~$ python
Python 3.7.4 (default, Aug 13 2019, 20:35:49)
[GCC 7.3.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 1 + 2
3
>>> a = 10
>>> b = 5
>>> a + b
15
>>> c = a + b
>>> print( c )
15
>>> c
15
>>> b = "Hi, there!"
>>> c = a + b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> for i in range( 5 ) :
...   print( i, i*i, i*i*i )
...
0 0 0
1 1 1
2 4 8
3 9 27
4 16 64
>>> quit()
(base) dalioba@achpiel:~$
```

Compilation vs. Interpretation

- In both cases, instructions are executed on the target processor.

- A compiler generates instructions by analyzing the source code in its *entirety* and optimizes the generated code based on the *entire source code* and specific optimizations.

- The generation of the instructions is (generally) independent of the execution of the target program.

Compilation vs. Interpretation

- An interpreter typically reads one “line” at a time.
  - It cannot perform overall analysis of the code as it does not know all of the code.
  - It therefore executes one line at a time without optimization.
- It must read the source code and generate instructions as it is executing the target program.
- It may take several operations for an interpreter to accomplish the same operation a compiler would have generated one machine instruction for.

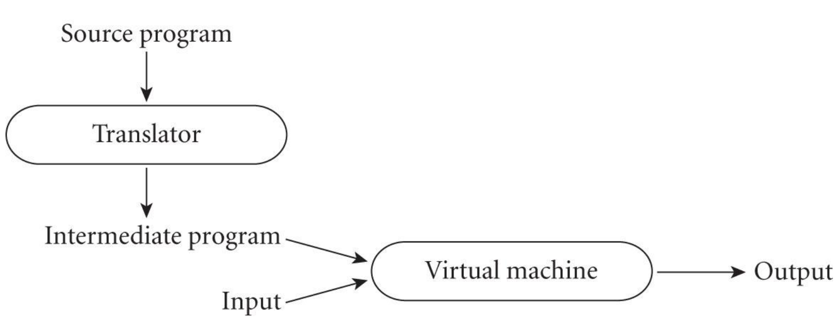
Compilation vs. Interpretation

- Interpretation
  - Don’t have to wait for compile and link steps.
  - Usually takes less space than a compiler.
  - Much easier to port an interpreter to a new architecture.

- Compilation
  - Generally much better performance.
  - Generally catches many errors earlier, before the target program even executes.

Compiling *and* Interpreting a Program

- While compilation and interpretation are different, they can be used together.



Compiling *and* Interpreting a Program

- If the initial translator is “simple”, we’d still call this interpretation. If it’s “complex” we’d call this compilation.
  - Subjective!
  - Also, both parts can be quite complex, as in the case of Java.
- Instead, we’ll use “compiling” to mean that a *complete analysis* of the source code is done by the translator.
  - Not just a “mechanical” transformation, as, e.g., cpp does.
- Also, the intermediate program would not bear a strong resemblance to the source code. It’s a *nontrivial* translation.

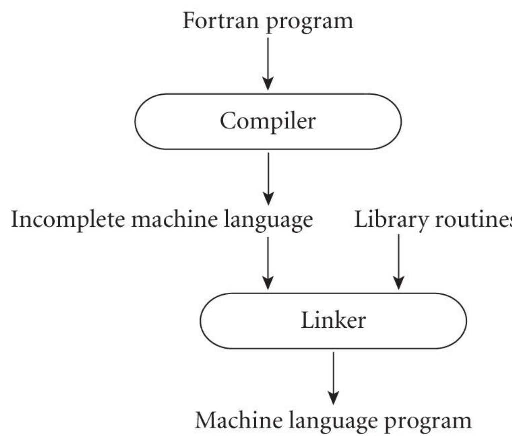
Pure Interpretation

- The earliest implementations of Basic were close to pure interpreters.
  - The original characters were read and reread and reread as the source code was interpreted. Removing comments sped up the program’s execution!

- Generally, a modern interpreter processes the source code to remove whitespace and comments, group characters into *tokens*, and even perhaps identify syntactic structures.

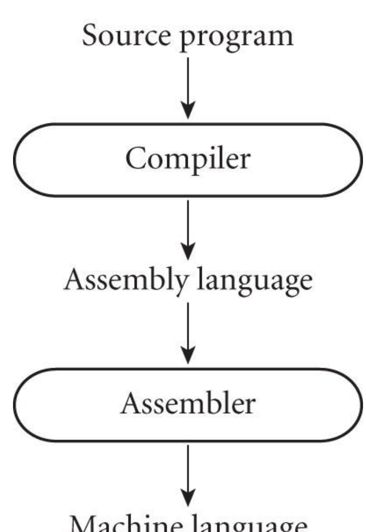
Pure Compilation

- (Early) Fortran implementations however were close to pure compilation.
  - The source code is translated into machine code.
  - May have a *library* of subroutines for shared use.
  - A *linker* puts it all together.
- There's still *some* interpretation.
  - `FORMAT` statements



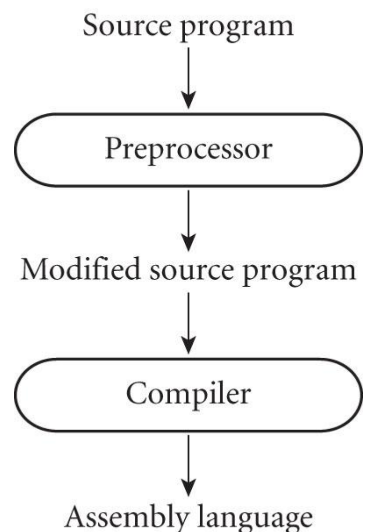
Chain of Compilation

- Many compilers generate assembly language instead of machine code.
  - Can make debugging easier
  - Helps isolate compiler from changes in the operating system



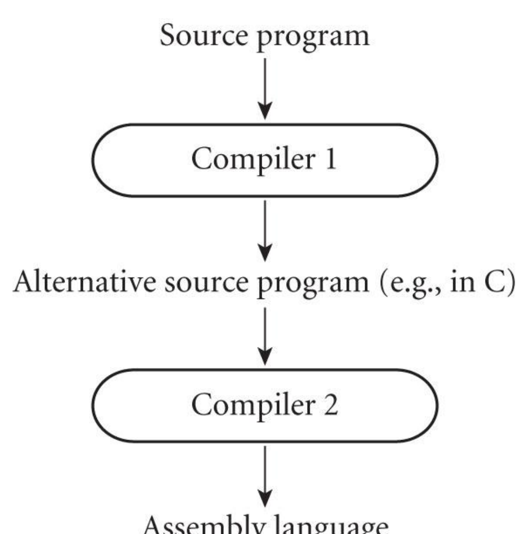
Chain of Compilation

- Many compilers (e.g., C) begin with a *textual* preprocessor that,
  - Removes comments
  - Expands *macros* (`#define`)
  - Provides *insertion* (`#include`)
  - Provides *conditional compilation* (`#if`)



Chain of Compilation

- Some compilers generate a relatively high-level intermediate program and then use *another* compiler to get to the final target program.
  - Called *source-to-source* compiling, C++ was originally implemented this way.
  - Still considered a true compiler!
    - Full analysis, non-trivial transform



Self-Hosting Compiler

- A *self-hosting* compiler is written in its own language.
  - An Ada compiler written in Ada, a C compiler written in C, and so forth.

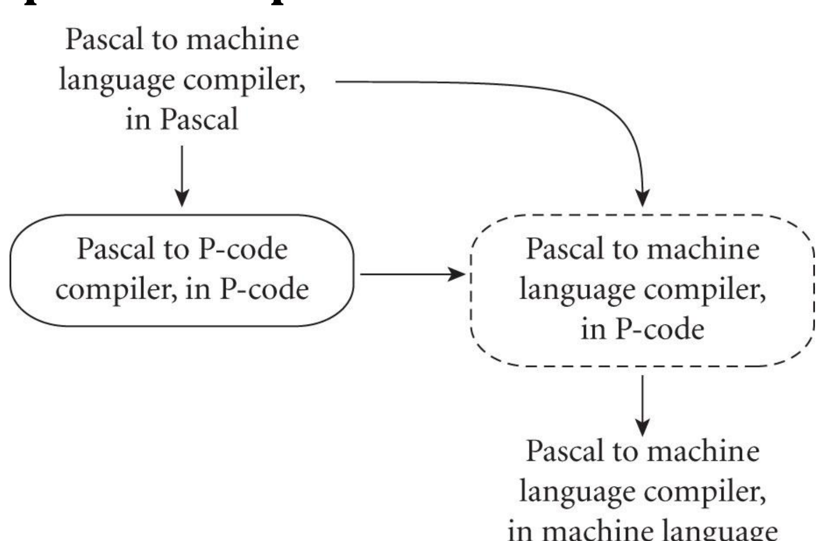
- So how to compile it the first time?
  - *Bootstrapping*: Start with a very simple implementation that knows just enough to get to the next level.
    - Often the earliest parts are *interpreters*.
  - Each step up adds more capability until the entire compiler can be compiled.

Self-Hosting Compiler Example

The original Pascal distribution included:

- A Pascal compiler, written in Pascal, that generates *P-code*.
- The same compiler, already in P-code.
- A P-code interpreter written in Pascal.

So, translate the P-code interpreter into a local language, then use it to run the P-code version of the compiler, then compile the Pascal version of the compiler.



[*P-Code*]

- A “P-Code” is a very simple language that is easy to translate to machine code.
  - “P” for *portable* or *pseudo*.
  - Used to make it easier to port software from one machine to another.
  - Also used to isolate compiler front ends from back ends.

- P-Code can be considered an intermediate form in the compilation process.

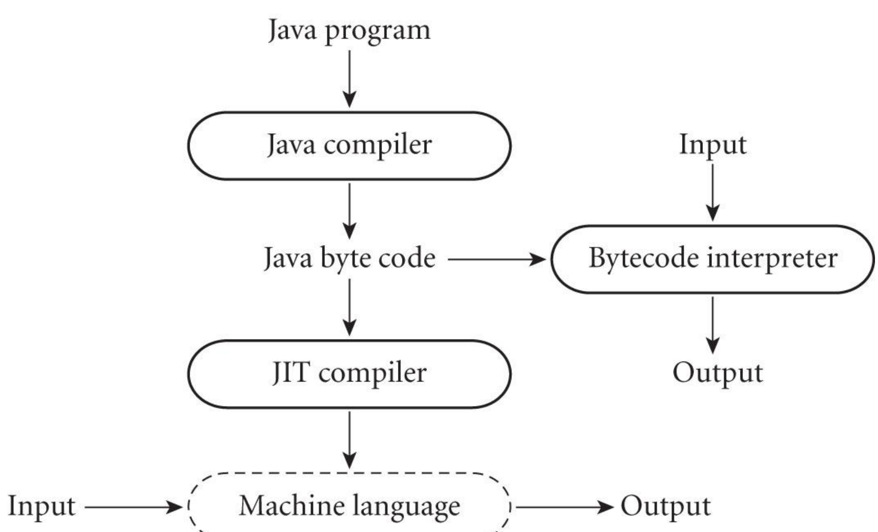
Compiling Interpreted Languages

- Some programs in traditionally interpreted languages can be compiled under a set of assumptions (about, e.g., types, bindings).
  - If at run-time the assumptions are violated, the program drops back into interpreted mode.
  - Otherwise, the performance advantages of compiled code can be obtained.

Just-In-Time Compilation

- An extension of this concept is *JIT*.
- Compilation can be *delayed* until the last possible moment.
- Allows for handling "on-the-fly" code, optimization based on run-time conditions, etc.
- Also supports platform-independent intermediate representation that can be distributed and then compiled locally into machine code.
- Examples include Java, C#.

Just-In-Time Compilation



Overview of Compilation

- Compilation is one of the most intensely studied aspects of computer science.

- Much of compilation has been studied so much that what used to be very hard is now fairly straightforward (not to say *easy*).
  - The first Fortran compiler cost 18 staff-years.
  - Now, writing a “compiler” is a semester project for an undergraduate.

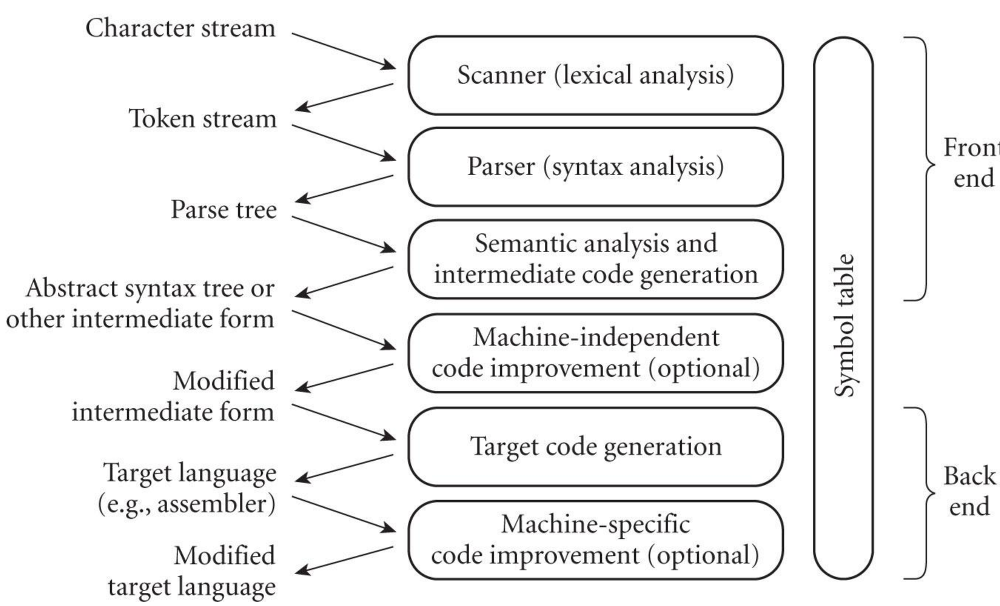
Overview of Compilation

- The compilation process is divided into *phases*
  - Each phase determines new information to be used later or transforms what has already been learned for later use.
  - The first phases analyze the source program to ***discover its meaning***. This is the compiler *Front End*.
  - The last phases ***construct*** (and possibly improve) ***the target program***. This is the compiler *Back End*.

Overview of Compilation

- One may hear of compiler *passes*
  - A *pass* is a phase or set of phases that are serialized with the respect to the others.
  - It does not start until all previous phases / passes have completed and it runs to completion before subsequent phases / passes start.
  - Sometimes a pass may be an entirely separate program, reading a description from and writing one to files.
  - Passes can be used to share parts of the compilation process.

Phases of Compilation



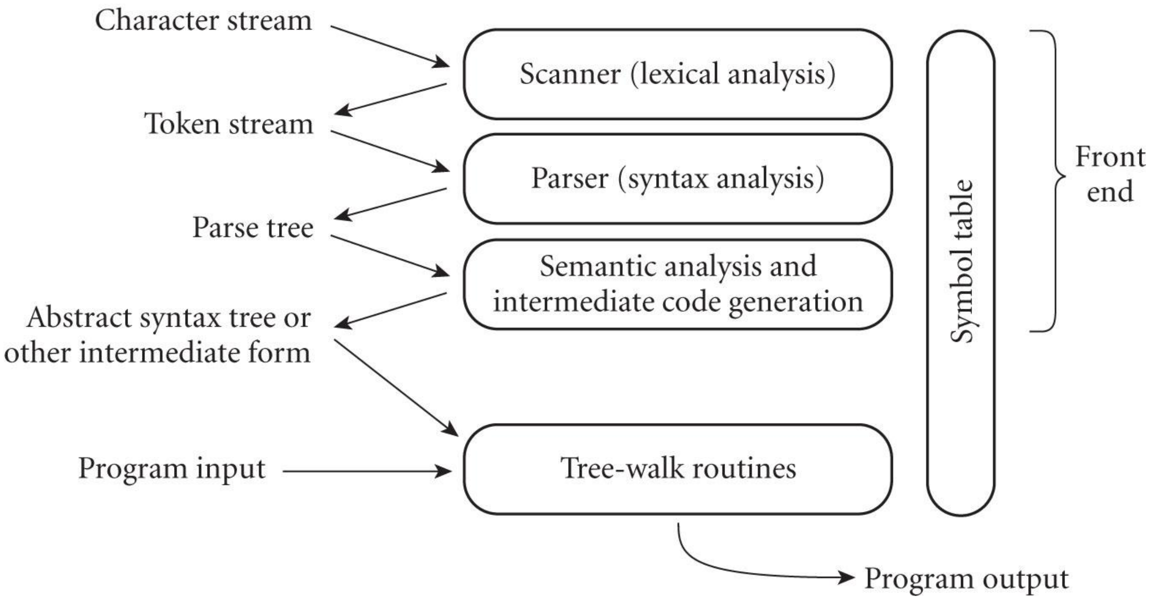
Front End vs Back End

- The *Front End* of a compiler (or interpreter) comprises the lexical, syntactic and semantic analysis steps along with the generation of intermediate code.
- The *Back End* of a compiler comprises the optimization of the intermediate code, target code generation, and machine-specific code optimization steps.
  - Interpreters don't have back ends.
    - (At least not the way compilers do.)
- The front end is concerned with determining what the program ***means***. The back end with ***producing the corresponding target code***.

Overview of Interpretation

- An interpreter shares the general structure of a compiler up through the Front End.
- However, instead of generating a target program, the interpreter "executes" the intermediate form directly.
  - This execution is commonly accomplished by a set of mutually-recursive routines that traverse ("walk") the abstract syntax tree, executing its nodes in order.
  - Another technique is to generate *bytecode* and hand it off to a bytecode engine. (This is getting closer to compilation.)

Phases of Interpretation



Overview of Compilation

- Lexical Analysis (*Scanning*)
    - The *scanner* (or *lexer*) reads individual characters and groups them into *tokens*.
    - These tokens are the smallest meaningful units of a program.

```
int main() {
  int i = getint(), j = getint();
  while (i != j) {
    if (i > j) i = i - j;
    else j = j - i;
  }
  putint(i);
}
```

```
int     main   (     )   {   int     i       =
getint  (      )     ,   j   =       getint  (
)       ;      while (   i   !=      j       )
{       if     (     i   >   j       )       i
=       i      -     j   ;   else    j       =
j       -      i     ;   }   putint  (       i
)       ;      }
```

Overview of Compilation

- Syntactic Analysis (*Parsing*)
  - Organizes the tokens into a *parse tree* that hierarchically represents higher-level constructs in terms of their lower-level parts.
  - Each construct is a *node*, its parts are its *children*, the *leaves* are the tokens from the lexical analysis phase.
  - The tree is constructed from the tokens by means of a set of recursive rules known as a *context-free grammar*.
  - The grammar defines the *syntax* of the language.

Overview of Compilation

- Context-free grammar excerpt, from C.
  - Shows the syntax of the `while` statement.
  - ε represents the empty string.
- Quite complicated!
- In general, parse trees have an insanely immense amount of information, even for relatively small programs.

iteration-statement ⟶ `while` (*expression*) *statement*

statement ⟶ *compound-statement*
compound-statement ⟶ { *block-item-list_opt* }

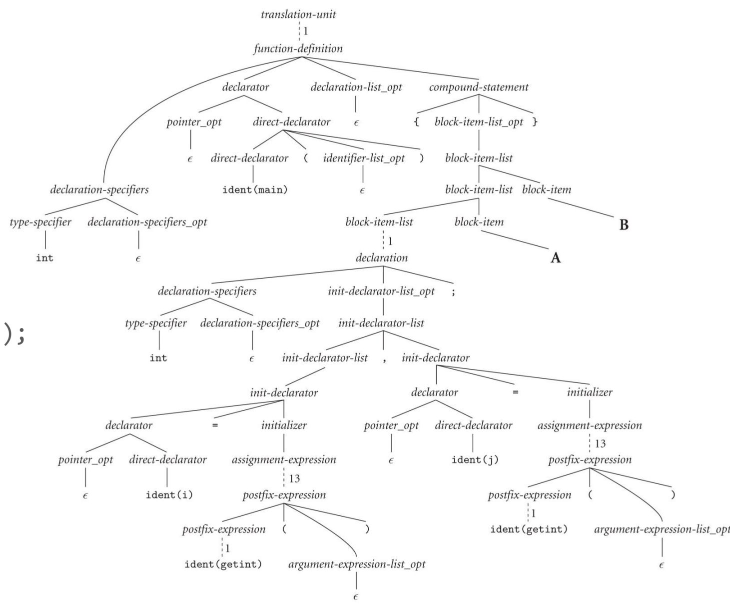
block-item-list_opt ⟶ *block-item-list*
block-item-list_opt ⟶ ε

block-item-list ⟶ *block-item*
block-item-list ⟶ *block-item-list block-item*
block-item ⟶ *declaration*
block-item ⟶ *statement*

http://www.quut.com/c/ANSI-C-grammar-y.html

Overview of Compilation

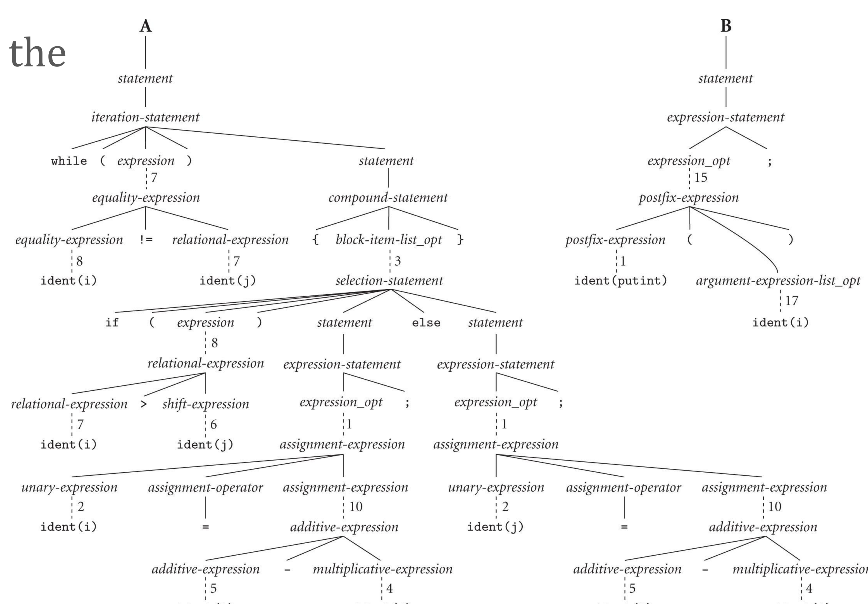
- Parse Tree for the GCD example (Part 1)




```
int main() {
  int i = getint(), j = getint();
  while (i != j) {
    if (i > j) i = i - j;
    else j = j - i;
  }
  putint(i);
}
```

Overview of Compilation

- Parse Tree for the GCD example (Part 2)



Overview of Compilation

- Semantic Analysis
  - Discovers the *meaning* of the source program. For example,
    - Determines when the use of the same identifier means the same program entity.
    - Ensures uses are consistent, both as a test of the legality of the source program and to guide generation of code in the back end.
  - Builds and maintains a *symbol table* to map each identifier to the information known about it, including
    - Type, internal structure (if any), scope, ...

Overview of Compilation

- Semantic Analysis
  - Enforces a large variety of rules that are not captured by the syntactic structure of the program, for example (in C),
    - Declaration of identifier before use
    - No use of identifier in an inappropriate context
    - Correct number / type of parameters in subroutine calls
    - Distinct, constant labels on the branches of a switch statement
    - Non-`void` return type function returns a value explicitly
  - These example rules are *static semantic* checks as they depend on only the structure of the program as it is written and can be enforced at compile time.

Overview of Compilation

- Semantic rules that can't be enforced until runtime are *dynamic semantics*, for example,
  - Variables are not used in an expression unless they have been assigned a value.
  - Pointers are not dereferenced unless they refer to a valid address.
  - Array subscripts are within bounds.
  - Arithmetic expressions do not overflow.

Overview of Compilation

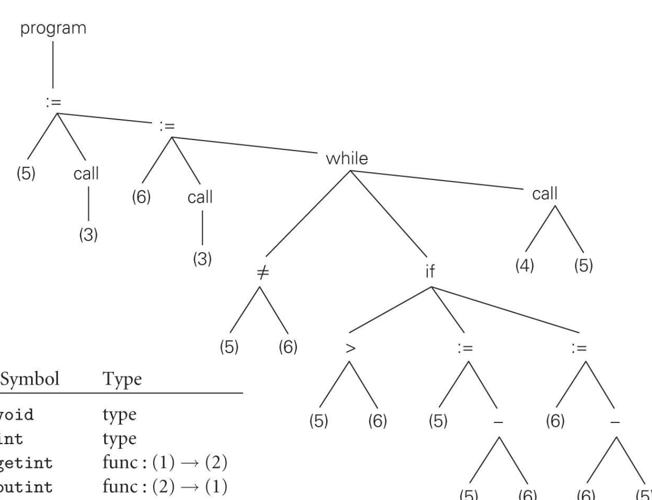
- Those dynamic semantics rules just given do *not* apply to C.
  - It would have saved a lot of trouble and debugging time if they were automatically enforced.
  - Q: Why aren't they?
- C has very little in the way of dynamic semantics rule enforcement.
  - As in … *none*. (Is that true?)

Overview of Compilation

- The parse tree is sometimes known as a *concrete syntax tree* because it completely shows how a sequence of tokens was derived using the CFG (Context-Free Grammar).
- As the static semantics phase runs, it transforms the parse tree to an *abstract syntax tree (AST)* which retains only the essential information.
- Annotations are also made with information useful to the rest of the process.
- Nodes get *attributes* added to them as required.

Overview of Compilation

- Abstract Syntax Tree and symbol table for GCD program




```
int main() {
  int i = getint(), j = getint();
  while (i != j) {
    if (i > j) i = i - j;
    else j = j - i;
  }
  putint(i);
}
```

Index	Symbol	Type
1	void	type
2	int	type
3	getint	func : (1) → (2)
4	putint	func : (2) → (1)
5	i	(2)
6	j	(2)

Overview of Compilation

- It's common for an interpreter to use the AST as its representation of the program to run.
- "Execution" then amounts to a traversal of the AST by a *tree-walker* routine that takes the appropriate action at each node.

[*Tree-Walker Functions as Interpreter*]

```
interpretStatementList( statementList, symbolTable )
  for statement in statementList
    switch statement.kind
      case ASSIGNMENT
        symbolTable[ statement.lvalue ].value =
          evaluateExpression( statement.rvalue, symbolTable  )

      case IF
        if evaluateExpression(
            statement.test, symbolTable  ) then
          interpretStatementList(
            statement.thenSide, symbolTable  )

        else
          interpretStatementList(
            statement.elseSide, symbolTable  )

      case WHILE
        while evaluateExpression(
          statement.test, symbolTable ) do
          interpretStatementList(
            statement.body, symbolTable )

      default
        print "WTF?  I don't understand", statement.kind,
          "as a statement kind."
```

```
evaluateExpression( expression, symbolTable )
  switch expression.kind
    case LITERAL
      return expression.value

    case BINARY_OPERATOR
      return performBOP(
        expression.operator,
        evaluateExpression( expression.left, symbolTable ),
        evaluateExpression( expression.right, symbolTable ) )

    case UNARY_OPERATOR
      return performUOP(
        expression.operator,
        evaluateExpression( expression.left, symbolTable ) )

    default
      print "WTF?  I don't understand", expression.kind,
        "as an expression kind."
      return 0
```

Overview of Compilation

- Many compilers use the AST as the *intermediate form (IF)* handed off to the back end for code generation.
- Others compilers *tree walk* the AST and generate a different intermediate form.
  - A common IF is a *control-flow graph*, whose nodes are fragments of assembly language for an idealized machine.

Overview of Compilation

- Target Code Generation
  - Translates the intermediate form into the target language.
  - Generating code that *works* is not all that hard.
  - Generating *good* code is trickier.

```
    pushl   %ebp            # \
    movl    %esp, %ebp      #  ) reserve space for local variables
    subl    $16, %esp       # /
    call    getint          # read
    movl    %eax, -8(%ebp)  # store i
    call    getint          # read
    movl    %eax, -12(%ebp) # store j
A:  movl    -8(%ebp), %edi  # load i
    movl    -12(%ebp), %ebx # load j
    cmpl    %ebx, %edi      # compare
    je      D               # jump if i == j
    movl    -8(%ebp), %edi  # load i
    movl    -12(%ebp), %ebx # load j
    cmpl    %ebx, %edi      # compare
    jle     B               # jump if i < j
    movl    -8(%ebp), %edi  # load i
    movl    -12(%ebp), %ebx # load j
    subl    %ebx, %edi      # i = i - j
    movl    %edi, -8(%ebp)  # store i
    jmp     C
B:  movl    -12(%ebp), %edi # load j
    movl    -8(%ebp), %ebx  # load i
    subl    %ebx, %edi      # j = j - i
    movl    %edi, -12(%ebp) # store j
C:  jmp     A
D:  movl    -8(%ebp), %ebx  # load i
    push    %ebx            # push i (pass to putint)
    call    putint          # write
    addl    $4, %esp        # pop i
    leave                   # deallocate space for local variables
    mov     $0, %eax        # exit status for program
    ret                     # return to operating system
```

Overview of Compilation

- Code Improvement
  - Often referred to as *optimization*, though that's a bit presumptuous.
  - Not required, but often done in an attempt to improve the generated code.
  - In this case, the optimizer did fairly well.
    - Got rid of most loads and stores as it was able to keep the values in the registers.

```
    pushl %ebp
    movl  %esp, %ebp
    pushl %ebx
    subl  $4, %esp
    andl  $-16, %esp
    call  getint
    movl  %eax, %ebx
    call  getint
    cmpl  %eax, %ebx
    je    C
A:  cmpl  %eax, %ebx
    jle   D
    subl  %eax, %ebx
B:  cmpl  %eax, %ebx
    jne   A
C:  movl  %ebx, (%esp)
    call  putint
    movl  -4(%ebp), %ebx
    leave
    ret
D:  subl  %ebx, %eax
    jmp   B
```

Overview of Compilation

- Code Improvement
  - The previous optimization was at the target language level.
  - Another place code improvement can happen is much earlier in the compilation process, right after semantic analysis.
  - Generally, the earlier an optimization can be made, the greater the effect (improvement, we hope) on the final target program.

UNIVERSITY OF TEXAS ARLINGTON

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING