Comp 204: Computer Systems and Their Implementation

Lecture 19: Introduction to Compilers

Today

- Compilers
  - Definition
  - Structure
  - Passes
  - Lexical Analysis
    - Symbol table
      - Access methods

Compilers

- Definition:
  - A compiler is a program which translates a high-level source program into a lower-level object program (target)

History

- Late 1940ies (post-von Neumann)
  - Programs were written in machine code
    - `C7 06 0000 0002` (move the number “2” to location 0000 (hex)
    - Highly complex, tedious and prone to error
- Assemblers appeared
  - Machine instructions given as mnemonics
    - `MOV X,2` (assuming X has the value 0000 (hex))
    - Greatly improved the speed and accuracy of writing code
    - But still non-trivial, and non-portable to new processors
- Needed a mathematical notation
  - Fortran appeared between 1954-57
    - `X = 2`
    - Exploited context free grammars (Chomsky) and finite state automatata…

Compiler

- Responsible for converting source code into executable code.
  - Analyses the code to determine the functionality
  - Synthesises executable code for a given processor
  - Optimises code to improve performance, or exploit specific processor instructions
- Assumes various data structures:
  - Tokens
    - Variables, language keywords, syntactic constructs etc
  - Symbol Table
    - Relates user defined entities (variables, methods, classes etc) with their associated values or internal structures
  - Literal Table
    - Stores constants, strings, etc. Used to reduce the size of the resulting code
  - Syntax/Parse Tree
    - The resulting structure formed through the analysis of the code
  - Intermediate Code
    - Intermediate representation between different phases of the compilation

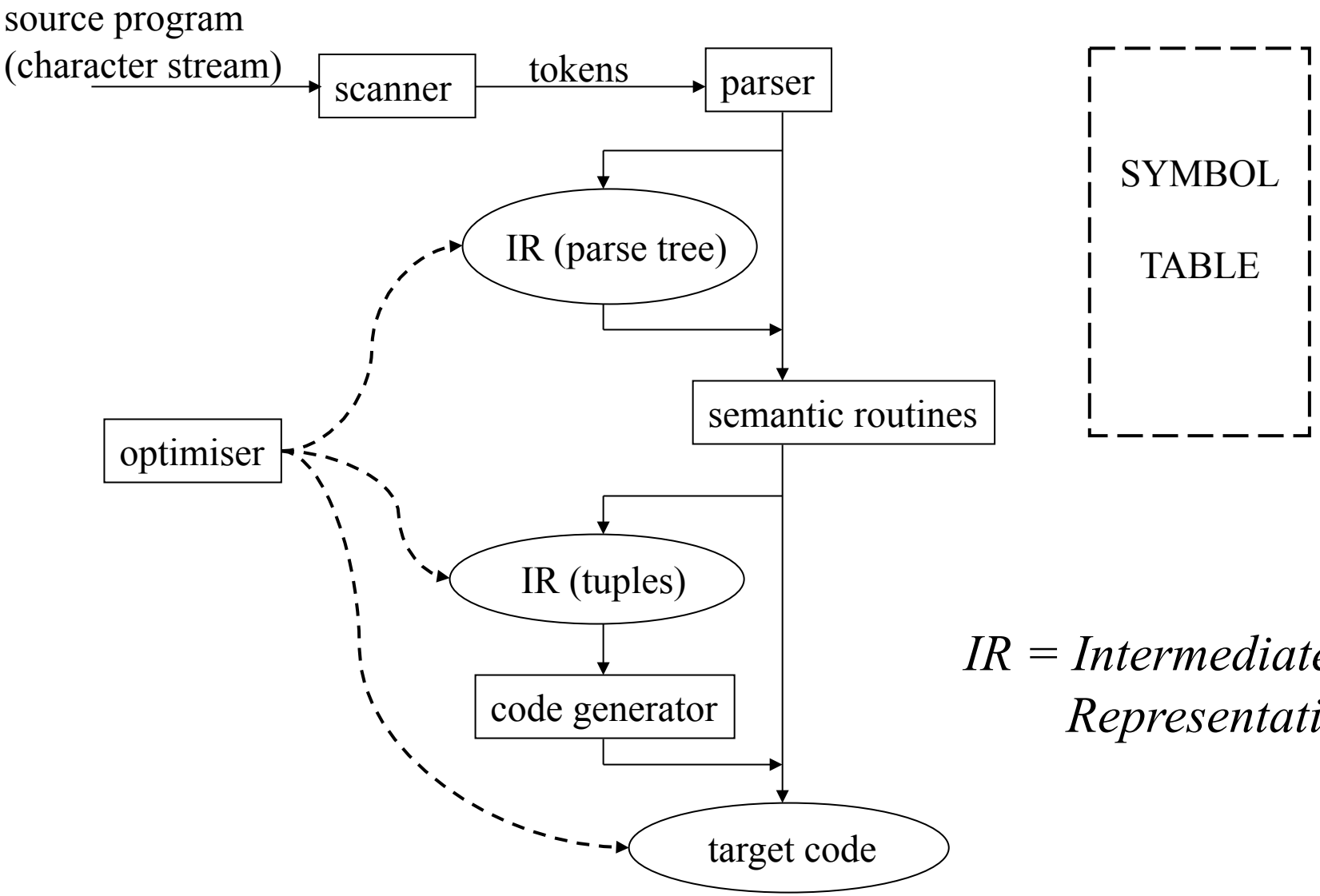
Phases and other tools

- **Interpreters:**
  - Unlike compilers, code is executed immediately
    - Slow execution, used more for scripting or functional languages
- **Assemblers:**
  - Constructs final machine code from processor specific Assembly code
    - Often used as last phase of a compilation process to produce binary executable.
- **Linkers:**
  - Collates separately compiled objects into a single file, including shared library objects or system calls.
- **Preprocessors:**
  - Called prior to the compilation process to perform macro substitutions
    - E.g. RATFOR preprocessor, or cpp for C code…
- **Profilers:**
  - Collects statistics about the behaviour of a program and can be used to improve the performance of the code.

Analysis and Synthesis

- Analysis:
  - checks that program constructs are legal and meaningful
  - builds up information about objects declared
- Synthesis:
  - takes analysed program and generates code necessary for its execution
- Compilation based on language definition, which comprises:
  - syntax
  - semantics

Compiler Structure



IR = Intermediate Representation

Compiler Organisation

- Each of compiler tasks described previously (in Compiler Structure) is a phase
- Phases can be organised into a number of passes
  - a pass consists of one or more phases acting on some representation of the complete program
  - representations produced between source and target are Intermediate Representations (IRs)

Single Pass Compilers

- One pass compilers very common because of their simplicity
- No IRs: all phases of compiler interleaved
- Compilation driven by parser
- Scanner acts as subroutine of parser, returning a token on each call
- As each phrase recognised by parser, it calls semantic routines to process declarations, check for semantic errors and generate code
- Code not as efficient as multi-pass

Multi-Pass Compilers

- Number of passes depends on number of IRs and on any optimisations
- Multi-pass allows complete separation of phases
  - more modular
  - easier to develop
  - more portable
- Main forms of IR:
  - Abstract Syntax Tree (AST)
  - Intermediate Code (IC)
    - Postfix
    - Tuples
    - Virtual Machine Code

Compiler Implementation

- Compilers often written in HLLs for ease of maintenance, portability, etc.
  - e.g. Pascal compiler written in C, runs on machine X
  - **Problem**: *always need both compilers available*
- To alter compiler:
  - Make necessary changes
  - Re-compile using C compiler
- To move to machine Y:
  - Re-write code generator to produce code for Y
  - Compile compiler on machine Y (using Y's C compiler)

Bootstrapping

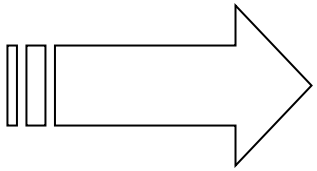
- Suppose our compiler is written in the language it compiles
    - e.g. C compiler written in C language
  - We can then run compiler through itself!
    - Bootstrapping
- To alter compiler:
  - Make necessary changes
  - Run compiler through itself
- To move to machine Y:
  - Re-write code generator to produce code for Y
  - Run compiler through itself to generate version of compiler that will run directly on Y

The Scanner (Lexical Analyser)

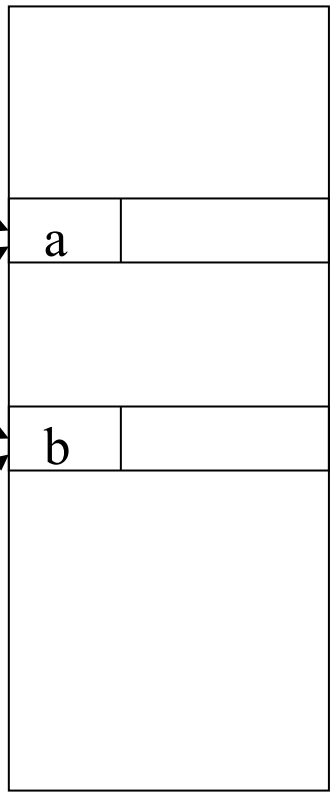
- Converts groups of characters into tokens (lexemes)
  - tokens usually represented as integers
  - white space and comments are skipped
- Each token may be accompanied by a value
  - could be a pointer to further information
- As identifiers encountered, entered into a symbol table
  - used to collect info. about declared objects
- Scanners often hand-coded for efficiency, but may be automatically generated (e.g. Lex)

Example

```
begin
  int a; float b;
  a = 1; b = 1.2;
  a = b + 1;
  print (a * 2);
end
```



	TOKEN	VALUE
`begin`	beginsymb	
`int a;`	intsymb	
	iden	@
	semisymb	
`float b;`	floatsymb	
	iden	@
	semisymb	
`a = 1;`	iden	@
	assignsymb	
	integer	1
	semisymb	
`b = 1.2;`	iden	@
	assignsymb	
	float	1.2



Symbol Table Access

- The symbol table is used by most compiler phases
  - Even used post-compilation (debugging)
- Structure of table and algorithms used can make difference between a slow and fast compiler
- Methods:
  - Sequential lookup
  - Binary chop and binary tree
  - Hash addressing
  - Hash chaining

Sequential Lookup

- Table is just a vector of names
- Search sequentially from beginning
- If name not found, add to end
- Advantages:
  - Very simple to implement
- Disadvantages:
  - Inefficient
  - For table with N names, requires N/2 comparisons on average
  - Can slow down a compiler by a factor of 10 or more

Binary Chop

- Keep names in alphabetical order
- To find name:
  - Compare with middle element to determine which half
  - Compare with middle element again to narrow down to quarter, etc.
- Advantage:
  - Much more efficient than sequential
  - $\log_2 N-1$ comparisons on average
- Disadvantage:
  - Adding a new name means shifting up every name above it

Question

- If the symbol table for a compiler is size 4096, how many comparisons on average need to be made when performing a lookup using the binary chop method?

  a) 2
  b) 11
  c) 12
  d) 16
  e) 31

Answer: b
11 – as there are $\log_2 N-1$ comparisons on average

Binary Tree

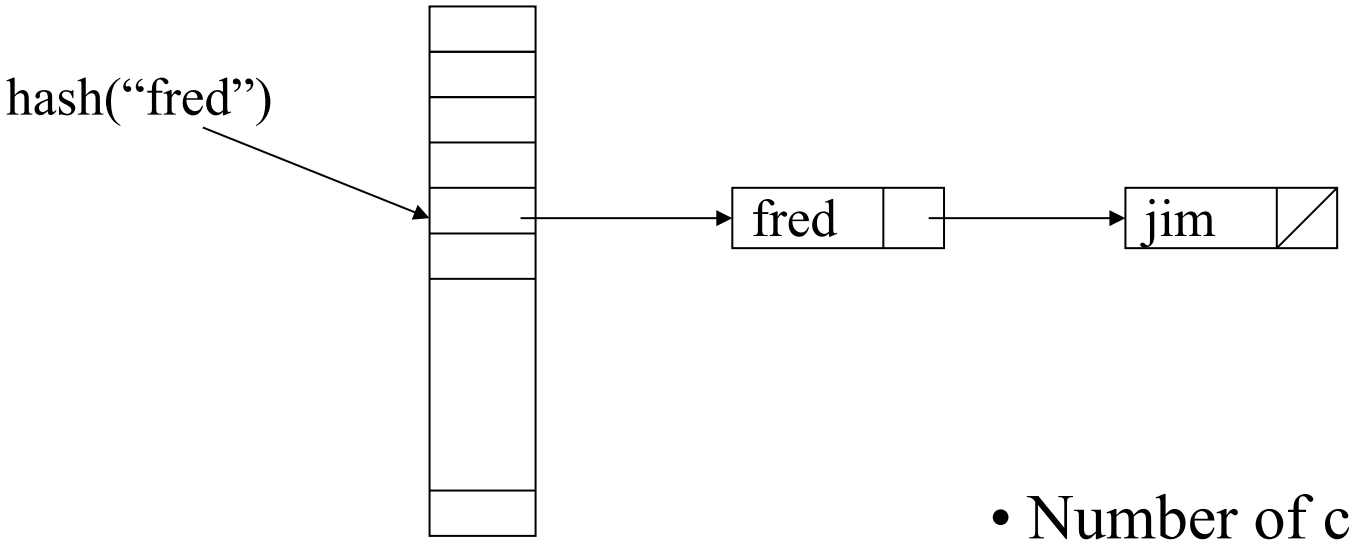
- Each node contains pointer to 2 sub-trees
  - Left sub-tree contains all names < current
  - Right sub-tree has all names >= current
- Advantages:
  - In best case, search time can be as good as binary chop
  - Adding a new name is simple and efficient
- Disadvantages:
  - Efficiency depends on how balanced the tree is
  - Tree can easily become unbalanced
  - In worst case, method as bad as sequential lookup!
  - May need to do costly re-balancing occasionally

Hash Addressing

- To determine position in table, apply a hash function, returning a hash key
  - Example fn: Sum of character codes modulo N, where N is table size (prime)
- Advantages:
  - Can be highly efficient
  - Even similar names can generate totally different hash keys
- Disadvantages:
  - Requires hash function producing good distribution
  - Possibility of collisions
  - May require re-hashing mechanism, possibly multiple times

Hash Chaining

- As before, but link together names having same hash key



array of pointers

• Number of comparisons needed very small

Question

- Concerning compilation, which of the following is NOT a method for symbol table access?

a) Sequential lookup
b) Direct lookup
c) Binary chop
d) Hash addressing
e) Hash chaining

Answer: b
Direct Lookup

Reserved Words

- Words like ‘for’, ‘while’, ‘if’, etc. are reserved words
- Could use binary chop on a table of reserved words first; if not there, search symbol table
- Simpler to pre-hash all reserved words into the symbol table and use one lookup mechanism