Contents

Introduction

- David Abrahamson
- david@cs.tcd.ie
- F16 ORI
- Examination
  - Coursework: 20%
    - 3 Graded Assignments
  - Exam: 80%

Labs

Labs	Date Given	Date Due
1	28 Sept	
2	4 Oct	

Labs	Date Given	Date Due
3	11 Oct	
4	18 Oct	
5	25 Oct	11 Nov
6	15 Nov	25 Nov
7	29 Nov	16 Dec

Language Processors

1. Interpreters: statements in a source language -> execution
2. Compilers/Translators: statements in a course language -> code in an object language
   - C++ -> C -> Machine Code
   - Java -> Machine Code

Translator

- Assemblers: low-level languages
  - one-to-one translation
- Compilers: high-level languages
  - one-to-many translation

Simple Model of a Compiler

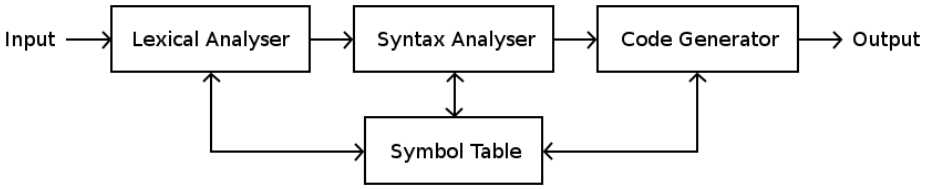

Figure 1: Simple Model of a Compiler

Symbol Table

- Stores information about various entities
- Variable `i`
- Variable name is in symbol table
- Value is **not** in symbol table
- Address in memory is in symbol table
- Type is in symbol table
- Modern compilers give access to symbol table when debugging

Lexical Analyser

Spits the input into lexical tokens.

Lexical Tokens:

- (class)
- (class, value)
- (class, pointer value)
  - Points to the symbol table

```
> if x>y then a:=b+c*d;
```

```
(if) (id, pointer x) (relational op, gt) (id, pointer y) (then)
(id, pointer a) (assign) (id, pointer b) (op, plus) (id, pointer c)
(op, times) (id, pointer d) (semicolon)
```

then, := are lexemes (lexical element) pointer x = symbol table entry to x whitespace and comments do not become lexical token

Syntax Analyser

- Verifies the input is syntactically correct
- Translates the sequence of lexical tokens into a sequence of atoms
  - Each of these atoms describes one of the operations to be performed when the program is running
  - The sequence of atoms should reflect the order in which the operations are to be performed at runtime
  - Multiple lexical tokens can become one atom

```
> A := B+C*D

(id, pointer A) (assign) (id, pointer B) (op, plus) (id, pointer C)
(op, times) (id, pointer D)

{MULT, pointer PR1, pointer C, pointer D}
{ADD, pointer PR2, pointer B, pointer PR1}
{ASSIGN, pointer A, pointer PR2}
```

> Assign as a lexical token is different to the atom assign Destination in leftmost pointer

Code Generator

- Expands the sequence of atoms into object code

```
LDR R5, C
LDR R6, D
MUL R5, R6

LDR R7, B
WHERE IS PARTIAL RESULT 1? R5!
ADD R7, R5

STR A, R7
```

> C = memory address allocated to it PR1 = destination where the partial result of the mult atom can be found, i.e. R5 PR2 = destination where the partial result of the add atom can be found, i.e. R7

Semantic Analysis

Source Code -> Lexical Analyser -> Semantic Analyser -> Syntax Analyser -> Code Generator -> Code Optimiser ->Object Code >————————>—->Symbol Table<—-<————<

- Processing related to the meaning of the symbols
  - What's the type of variable A? Variable B?
  - If one is a float and another an int, float must be converted to an int to perform an ADD
- Can do static semantic analysis inside of syntax analyser
- Code generator can do dynamic semantic analysis

- Dave Abe thinks this isn't necessary

Don't feel contrained by phases. Do processing when its most convenient.

- `lookup (I, TAB)`
  - If you already know in the lexical analyser it's a function call, don't wait until syntax analyser

Code Optimisation

- Just code improvement techniques
  - Compilers 2 is almost exclusively about code optimisation

One-Pass System

- Lexical Analyser passes lexemes into syntax analyser
- Syntax analyser keeps getting input till it recognises syntax
- Passes produced sequence of atoms into code generator

Multi-Pass

- System does one pass to get something (i.e. function declerations)
- Second pass to link
- etc.

Run Time Implementation

Describes the Data Structures and Control Mechanisms that exist when the program is running

- The Stack
- Arrays
  - Row/Column Major Order defined by language
- Symbol Table Management
  - How is it maintained?