CA4003 - Compiler Construction

Introduction

Dr. David Sinclair

Overview

This module will cover the compilation process, reading and parsing a structured language, storing it in an appropriate data structure, analysing the data structure and generating an executable program.

In this module we will cover:

- Structure of a compiler
- Lexical analysis
- Parsing
- Abstract syntax
- Semantic Analysis
- Intermediate Code Generation
- Register Allocation & Code Optimisation
- Run-time Environments
- Code Generation

Overview [2]

The obvious application of these techniques will be in compiling a high-level computer program into an executable program.

However these techniques can also be used to process and analyse any structured data.

Texts

Essential:

- Andrew W. Appel, *Modern Compiler Implementation in Java*, Cambridge University Press, 1998, ISBN 0-521-58388-8

Supplementary:

- Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D. Ullman, *Compilers: Principles, Techniques and Tools*, 2^{nd} Edition, Addison-Wesley, 2007, ISBN 0-321-49169-6

Contact Details

Lecturer: Dr. David Sinclair
Office: L253
Phone: 5510
Email: David.Sinclair@computing.dcu.ie
WWW: https://www.computing.dcu.ie/∼davids

Course web page:
https://www.computing.dcu.ie/∼davids/courses/CA4003/CA4003.html

How do I successfully complete this module?

The module mark is a straight weighted average of:

- 30% continuous assessment
  - 2 assignments
    - first assignment: Front-end - lexical, syntax analysis (15%)
    - second assignment: Back-end - semantic analysis and generating intermediate code (15%)
- 70% end-of-semester examination
  - 10 questions. Do all questions.

Compiler and Interpreters

What is a compiler?

- It is a program that translates an *executable* program in one language into an *executable* program in another language.

What is an interpreter?

- It is a program that reads an *executable* program and produces the results of running that program.
- This typically involves executing (or evaluating) the source program.

Many of the same front-end issues arise in *interpreters* and *compilers*.

Overview of the Compilation Process

The compilation process consists of a number of *phases*. The number of *phases* varies from compiler to compiler depending on their complexity.

A basic set of *phases* are:

- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- Intermediate Code Generation
- Code Optimisation
- Code Generation

The first 3 *phases* comprise the front-end of the compiler.

The second 3 phases comprise the back-end of the compiler.

Lexical Analysis

The goal of *lexical analysis* is to convert a stream of characters from the source program into stream of *tokens* that represent recognised keywords, identifiers, numbers and punctuation.

Some tokens, such as identifiers and numbers, require an additional quantity, called a *lexeme*, that indicates the type and/or value of the token.

$$answer = x * 2 - y$$
$$\downarrow$$
id($answer$), =, id(x), $*$, num(2), -, id(y)

Syntax Analysis

The goal of the *syntax analysis* is to combine the *tokens* generated by the *lexical analysis* into a valid "sentence".
A *grammar* is a set of rules that specifies how the *tokens* can be combined.

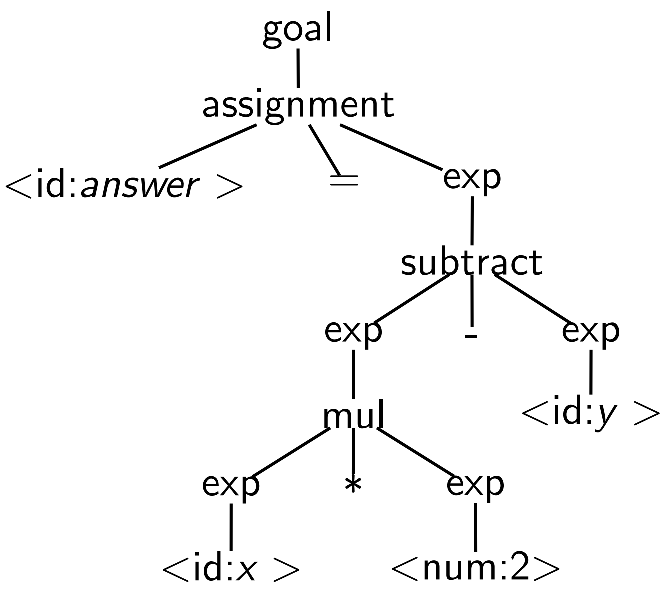
Let's assume the following grammar.

goal	→	assignment | exp
assignment	→	id '=' exp
exp	→	id | num | mul | add | subtract
mul	→	exp '*' exp
add	→	exp '+' exp
subtract	→	exp '-' exp

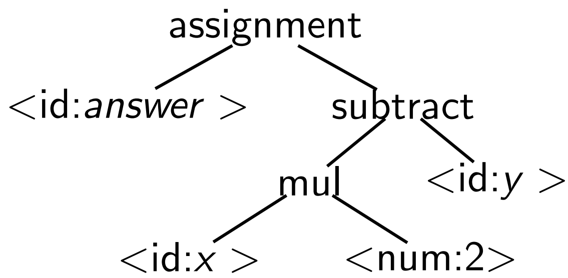
The above grammar is very flawed. We will see why later in the course.

Syntax Analysis [2]

Parse Tree



Abstract Syntax Tree (AST)



The Abstract Syntax Tree (AST) is a compressed version of the Parse Tree, but without the redundant information.

Semantic Analysis

In a compiler this phase checks the source program for semantic errors, and gathers type information for the intermediate code generation phase.

For example, what if *answer* and *y* are integer and *x* is a float?

In an interpreter this phase evaluates the source program stored in the AST.

Intermediate Code Generation

Intermediate code is a kind of abstract machine code which does not rely on a particular target machine by specifying the registers or memory locations to be used for each operation.

This separates compilation into a mostly language dependent *front end*, and a mostly machine-dependent *back end*.

For example:

```
loop: JLE x 0 end
      SUB x 1 temp
      MOV temp x
      JMP loop
end:  ...
```


Code Optimisation

This is an optional phase which can be used to improve the intermediate code to make it run faster and/or use less memory.

For example, the variable `temp` in the previous fragment of intermediate code is not required. This can be removed to give the following:

```
loop: JLE x 0 end
      SUB x 1 x
      JMP loop
end:  ...
```

Code Generation

This phase translates intermediate code into object code, allocating memory locations for data, and selecting registers.

This can also include a *linking phase* when the language allows the source code to be written in separate files.