EDAN65: Compilers - Reference Sheet

Karl Hallsby

Last Edited: September 3, 2019

Contents

1 Introduction

There are numerous steps in the compilation process of a standard program. Each phase converts the program from one representation to another.

1. Lexical Analysis/Scanning
2. Syntactic Analysis (Parsing)
3. Semantic Analysis
4. Intermediate Code Generation
5. Optimization
6. Target Code Generation

Defn 1 (Syntactic Analysis (Parsing))**.** *Syntactic Analysis* or *Parsing* is the process where tokens are input and an AST (Abstract Syntax Tree) is created. This AST is generated based on the input source code and the Lexical Analysis (Scanning) that occurs.

This code would generate an error during the Syntactic Analysis (Parsing).

```
int r( {
  return 3;
}
```

This wouldn't fail during Lexical Analysis (Scanning) because the scanner doesn't care that the parentheses don't match. All that it cares about is that there are parentheses that it needs to mark. During the Syntactic Analysis (Parsing) we find out that the syntax would be wrong. This would happen because we can't line our tokens up correctly in our AST.

Remark 1.1. Syntactic Analysis (Parsing) *ONLY* handles the reading in of tokens and creating an Abstract Syntax Tree. It *DOES NOT* attach any meaning to anything. Therefore, this does not return an error during Syntactic Analysis (Parsing).

```
integer q() {
  return 3;
}
```

However, it does return an error during Semantic Analysis.

Defn 2 (Semantic Analysis)**.** *Semantic Analysis* is the phase of the compilation process that takes the AST (Abstract Syntax Tree) and attaches some semblance of meaning to the tokens in the tree. We determine what each "phrase" means, relate the uses of variables to their definitions, check types of expressions, and request translations of each "phrase". This is the point in the compilation process where the strings that were read in by the scanner and organized by the parser have any meaning. Before this, the only things that can be caught are token errors, and the like. So, this will generate an error that is caught during Semantic Analysis.

```
integer q() {
  return 3;
}
```

Because `integer` isn't a valid keyword in the Java language, at least not by default, and not capitalized like that, it gets caught during Semantic Analysis.

This would also generate an error during Semantic Analysis.

```
int p(int x) {
  int y;
  y = x * 2;
}
```

Both of these wouldn't be caught before the Semantic Analysis because the tokens read in during Lexical Analysis (Scanning) and organized during Syntactic Analysis (Parsing) do not have any meaning any earlier.

2 Lexical Analysis/Scanning

Defn 3 (Lexical Analysis (Scanning))**.** *Lexical Analysis* or *Scanning* is the phase of the compilation process that reads in the source code text. It breaks the things it reads into *tokens*.

Remark 3.1. Lexical Analysis (Scanning) *ONLY* handles the reading IN of source code and the outputting of tokens. It *DOES NOT* attach any meaning or put anything together.

This means that these are the ***ONLY*** types of errors that will be caught.

```
1  int #s() {
2    return 3;
3  }
```

Because the # token isn't understood by the scanner, the whole thing fails. The Scanner is just a simple look up device. It can only find things that it knows about. If it sees something that it has no clue about, it fails.

There are several ways to implement a scanner. One of the most common ways is the use of a Finite State Automaton or Finite State Machine.

Defn 4 (Finite State Automaton)**.** A *finite state automaton* or *finite state machine* is a mathematical model of computation. It is an abstract machine that can be in exactly one of a finite number of states at any given time. The FSM can change from one state to another in response to some external inputs; the change from one state to another is called a transition. An FSM is defined by a list of its states, its initial state, and the conditions for each transition.

There are 2 types of finite statue automata:

1. Deterministic Finite State Automata
2. Non-deterministic Finite State Automata

A deterministic finite state automata can be constructed to be equivalent to any non-deterministic one.