1 Parsing

1.1 Parsing Abstracted

Often when scanning or parsing, we relate the process to forming words and sentences in natural language. Just as scanning produces tokens which are akin to forming letters into words, parsing is much like using these words to form more complex sentences, which allows us to derive additional meaning from our programs.

We have already seen that our scanner can produce tokens based on what sequence of characters are scanned. Parsing is much the same, the only difference is that instead of producing tokens, we produce terminal and non terminal elements. An important consideration we have to make is what kind of grammar we wish to parse. Different grammars require varying degrees of effort to parse, and some grammars are unable to represent certain programming languages.

2 Types of Grammars

2.1 CFGs

***CFGs* are sentences that we consider valid for a certain programming language. *Context free grammars* for our purposes can be broken up into two types.**

2.1.1 LL Grammars

LL grammars are easy to write homemade parsers for, but not all programming languages can be expressed with them. LL grammars are simple in that it only needs to consider the current rule and the next token in the stream, which is where its lack of universiality in langauge representation stems from. Languages and their grammars must be designed with intent in order to fit the limitations of LL grammars. If we want to create a homemade parser for a LL grammar, we should use a *recursive descent parser*.

2.1.2 LR Grammars

LR grammars are useful in that nearly all programming languages can be represented by them. The only issue is that writing a parser like this can be quite complex, and usually a parser generator is used like bison or some equivilant. These generatators take in as input an LR grammar and produce as output a parser.

3 Elements of a CFG

3.1 Terminals

Terminals are equivilant to tokens from our scanner. They are basic elements like identifiers, operators, and keywords. These can be represented as the set of lowercase letters [a-z].

- Identifiers.
- Operators.
- Keywords.

3.2 Non-Terminals

A non-terminal represents a valid structure in a language, but as its name suggests, is not a terminal. That is, it is neither a identifier, operator or keyword. Non-terminal elements typically include declarations, statements, and expressions. Non-terminals will be represented by the set of all uppercase letters [A-Z].

- Declarations.
- Statements.
- Expressions.

3.3 Sentences and Sentential Forms

A sentence is a valid sequence of terminals while a sentential form is a valid sequence of terminals and non-terminals. We will use the set of all Greek symbols to represent sentential forms $[\alpha - \Omega]$