Computer Science 340
Programming Languages
Siena College
Fall 2023

Topic Notes: Lexical Analysis

Lexical Analysis

We next consider *lexical analysis* – the process of identifying the small-scale language constructs.

Here, we identify the lexemes – names, operators, numeric literals, punctuation, line numbers (BASIC), etc.

In many ways, lexical analysis is similar to syntax analysis, but it is generally a easier problem.

Lexical analysis is usually performed separately from syntax analysis. Why?

- Simplicity: simpler approaches are suitable for lexical analysis
- Efficiency: focuses optimization efforts on lexical analysis and syntax analysis separately
- Portability: a lexical analyzer might not always be portable (due to file I/O), whereas syntax analyzer may remain portable

The lexical analyzer is typically a *pattern matcher*.

- Identifies and isolates lexemes
- Is a "front-end" for the parser, which can then deal strictly with tokenized input
- Lexemes are logical substrings of the source program that belong together
- Lexical analyzer assigns codes called tokens to the lexemes
  - *e.g.*, For a variable name `sum`, `sum` is a lexeme; and `IDENT` is the token

Before we look at specifics of how a lexical analyzer works, let's think about what some of these lexemes look like.

First, consider integer constants in C/C++. These include:

- an optional unary minus sign
- digits
- optional e notation

- different prefixes for octal and hexadecimal

`https://github.com/SienaCSISProgLang/intliterals`

To create a formal definition of an integer with the restriction that it must be in base 10 and that it does not use e notation:

$$(\epsilon \bigcup -)\cdot(1 \bigcup 2 \bigcup 3 \bigcup 4 \bigcup 5 \bigcup 6 \bigcup 7 \bigcup 8 \bigcup 9)\cdot(0 \bigcup 1 \bigcup 2 \bigcup 3 \bigcup 4 \bigcup 5 \bigcup 6 \bigcup 7 \bigcup 8 \bigcup 9)*$$

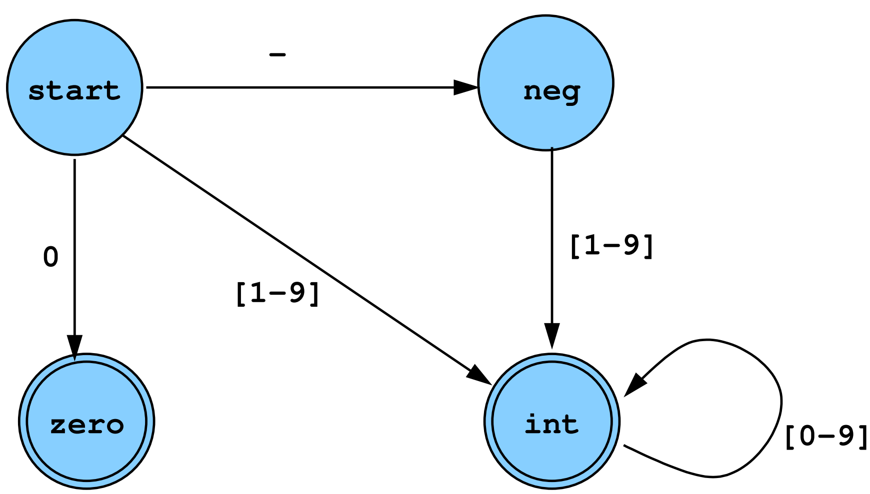
This means it's either nothing or a unary $-$, followed by one digit in the 1-9 range, then 0 or more copies of digits 0-9. The "0 or more copies of" is indicated by the $*$ at the end.

Alternately, we could use a Unix-like *regular expression*:

```
(-?[1-9][0-9]*|0)
```

Again, an optional $-$, one digit 1-9, zero or more digits 0-9, OR the whole thing can be a single 0.

We can also see this as a *deterministic finite automaton (DFA)* or *state diagram*.



This can also be described by a grammar.

```
<int-literal> => -<unsigned-int>
               | <unsigned-int>
               | 0
<unsigned-int> => [1-9]
                | [1-9]<one-or-more-digits>
<one-or-more-digits> => [0-9]
                      | [0-9]<one-or-more-digits>
```

A language is *regular* if

- It can be represented by a regular expression.
- It can be represented by a deterministic finite automaton (DFA).
- It can be represented by a regular grammar.

These are all equivalent statements.

We have seen grammars. A *regular grammar* is one that has a very restricted form for its productions:

- a production's right hand side (RHS) may be a single terminal
- a production's RHS may be a single terminal followed by a single nonterminal

A grammar is regular if and only if it produces a regular language.

The grammar given above for integer literals is not a valid regular grammar because of the second rule (its RHS is a single nonterminal). We can rewrite it a bit to eliminate this.

```
<int-literal> => -<unsigned-int>
               | [1-9]
               | [1-9]<one-or-more-digits>
               | 0
<unsigned-int> => [1-9]
                | [1-9]<one-or-more-digits>
<one-or-more-digits> => [0-9]
                      | [0-9]<one-or-more-digits>
```

We've basically put a copy of the productions for `<unsigned-int>` into the productions for `<int-literal>` to come up with an equivalent grammar which now does satisfy the requirements for a regular grammar.

A Lexical Analyzer

Our textbook has a demonstration of a simple lexical analysis program for arithmetic expressions in Section 4.2.

The best way to understand lexical analysis is to understand the relation between the state diagram below (from Sebesta) and a grammar, with a lexical analysis program, and to understand how the program works.

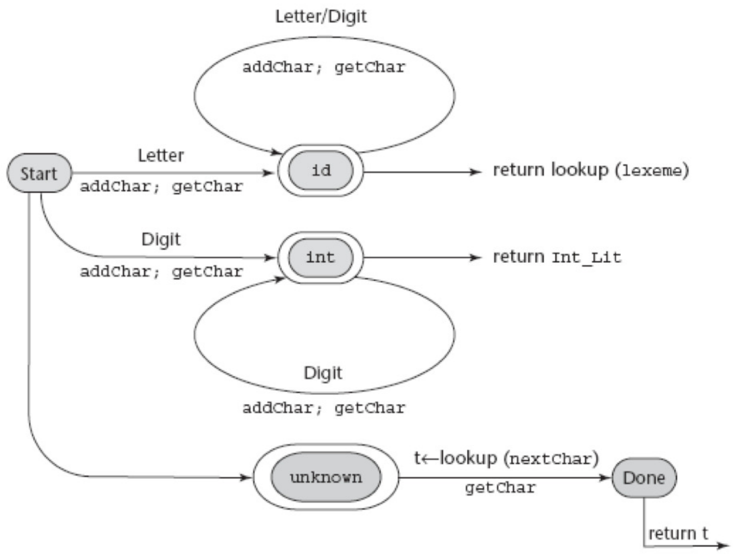


Figure 4.1 from Sebesta 2012.

An improved version of the C program from the text:

`https://github.com/SienaCSISProgLang/sebesta-lex`

See the extended comments in the code for more details.