## The Basics

*Lexical analysis* or *scanning* is the process where the stream of characters making up the source program is read from left-to-right and grouped into tokens. *Tokens* are sequences of characters with a collective meaning. There are usually only a small number of tokens for a programming language: constants (integer, double, char, string, etc.), operators (arithmetic, relational, logical), punctuation, and reserved words.

stream =...

while (i > 0)

i = i

-

;

2

Lexical

Analyzer

error messages

source

language

token

while

(

i

>

0

)

i

The lexical analyzer takes a source program as input, and produces a stream of tokens as output. The lexical analyzer might recognize particular instances of tokens such as:

## Scanner Implementation 1: Loop and Switch

There are two primary methods for implementing a scanner. The first is a program that is hard-coded to perform the scanning tasks. The second uses regular expression and finite automata theory to model the scanning process.

A "loop & switch" implementation consists of a **main** loop that reads characters one by one from the input file and uses a **switch** statement to process the character(s) just read. The output is a list of tokens and lexemes from the source program. The following program fragment shows a skeletal implementation of a simple loop and switch scanner. The main program calls **InitScanner** and loops calling **ScanOneToken** until **EOF**. **ScanOneToken** reads the next character from the file and **switch**es off that **char** to decide how to handle what is coming up next in the file. The return values from the scanner can be passed on to the parser in the next phase.

struct token\_t {

 int type; // one of the token codes from above union { char stringValue[256]; // holds lexeme value if string/identifier int intValue; // holds lexeme value if integer double doubleValue; // holds lexeme value if double

 } val;

};

 int main(int argc, char \*argv[])

{ struct token\_t token;

 InitScanner();

 while (ScanOneToken(stdin, &token) != T\_END) ; // here is where you would process each token return 0;

}

static void InitScanner()

{

 create\_reserved\_table(); // table maps reserved words to token type insert\_reserved("WHILE", T\_WHILE) insert\_reserved("IF", T\_IF) insert\_reserved("RETURN", T\_RETURN) ....

}

// ASCII value used as token type

The mythical source language tokenized by the above scanner requires that reserved words be in all upper case and identifiers in all lower case. This convenient feature makes it easy for the scanner to choose which path to pursue after reading just one character. It is sometimes necessary to design the scanner to "look ahead" before deciding what path to follow— notice the handling for the **'/'** character which peeks at the next character to check whether the first slash is followed by another slash or star which indicates the beginning of a comment. If not, the extra character is pushed back onto the input stream and the token is interpreted as the single char operator for division.

Loop-and-switch scanners are sometimes called *ad hoc* scanners, indicating their design and purpose of solving a specific instance rather a general problem. For a sufficiently reasonable set of token types, a hand coded, loop and switch scanner might be all that’s needed— it requires no other tools. The **gcc** front-end uses an ad hoc scanner, in fact. On the other hand, **gcc**’s C lexer is over 2,500 lines of code; verifying that such an amount of code is correct is much harder if your lexer does not see the extent of use that gcc’s front-end experiences.

## Scanner Implementation 2: Regular Expressions and Finite Automata

The other method of implementing a scanner is using regular expressions and finite automata. A quick detour for some background review and then let’s see how we can generate a scanner building on techniques from automata theory.