Lecture 10

Running the Scanner

Here is the output of the scanner when executed and given the file `main.cpp` as input, i.e., the scanner is being asked to provide tokens found in the file `main.cpp`:

`lex <main.cpp`

```
259,void
258,main
283,(
284,)
285,{
258,FlexLexer
258,lex
290,;
260,int
258,tc
266,=
258,lex
291,.
258,yylex
283,(
284,)
290,;
263,while
283,(
258,tc
276,!=
257,0
284,)
258,cout
279,<<
258,tc
279,<<
292,","
279,<<
258,lex
291,.
258,YYText
283,(
284,)
279,<<
258,endl
290,;
258,tc
266,=
258,lex
291,.
258,yylex
283,(
284,)
290,;
286,}
```

Flex input for C++

As an illustration of the power of Flex, here is the input for generating scanner for C++ compiler.

```
/*
 * ISO C++ lexical analyzer.
 * Based on the ISO C++ draft standard of December '96.
 */

%{
#include <ctype.h>
#include <stdio.h>
#include “tokdefs.h"

int lineno;

static int yywrap(void);
static void skip_until_eol(void);
static void skip_comment(void);
static int check_identifier(const char *);
%}

intsuffix    ([uU][lL]?)|([lL][uU]?)
fracconst    ([0-9]*\.[0-9]+)|([0-9]+\.)
exppart            [eE][-+]?[0-9]+
floatsuffix  [fFlL]
chartext     ([^'])|(\\.)
stringtext   ([^"])|(\\.)
%%
%%
"\n"     { ++lineno; }
[\t\f\v\r ]+    { /* Ignore whitespace. */ }

"/*"     { skip_comment(); }
"//"     { skip_until_eol(); }

"{"     { return '{'; }
"<%"     { return '{'; }
"}"     { return '}'; }
"%>"     { return '}'; }
"["     { return '['; }
"<:"     { return '['; }
"]"     { return ']'; }
":>"     { return ']'; }
"("     { return '('; }
")"     { return ')'; }
";"     { return ';'; }
":"     { return ':'; }
"..."     { return ELLIPSIS; }
"?"     { return '?'; }
"::"     { return COLONCOLON; }
"."     { return '.'; }
".*"     { return DOTSTAR; }
"+"     { return '+'; }
"-"     { return '-'; }
"*"     { return '*'; }
"/"     { return '/'; }
"%"     { return '%'; }
"^"     { return '^'; }
"xor"     { return '^'; }
"&"     { return '&'; }
```

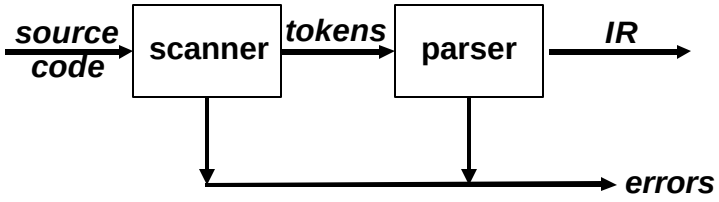
```
"bitand"    { return '&'; }
"|"     { return '|'; }
"bitor"     { return '|'; }
"~"     { return '~'; }
"compl"     { return '~'; }
"!"     { return '!'; }
"not"     { return '!'; }
"="     { return '='; }
"<"     { return '<'; }
">"     { return '>'; }
"+="     { return ADDEQ; }
"-="     { return SUBEQ; }
"*="     { return MULEQ; }
"/="     { return DIVEQ; }
"%="     { return MODEQ; }
"^="     { return XOREQ; }
"xor_eq"    { return XOREQ; }
"&="     { return ANDEQ; }
"and_eq"    { return ANDEQ; }
"|="     { return OREQ; }
"or_eq"     { return OREQ; }
"<<"     { return SL; }
">>"     { return SR; }
"<<="     { return SLEQ; }
">>="     { return SREQ; }
"=="     { return EQ; }
"!="     { return NOTEQ; }
"not_eq"    { return NOTEQ; }
"<="     { return LTEQ; }
">="     { return GTEQ; }
"&&"     { return ANDAND; }
"and"     { return ANDAND; }
"||"     { return OROR; }
"or"     { return OROR; }
"++"     { return PLUSPLUS; }
"--"     { return MINUSMINUS; }
","     { return ','; }
"->*"     { return ARROWSTAR; }
"->"     { return ARROW; }
"asm"     { return ASM; }
"auto"     { return AUTO; }
"bool"     { return BOOL; }
"break"     { return BREAK; }
"case"     { return CASE; }
"catch"     { return CATCH; }
"char"     { return CHAR; }
"class"     { return CLASS; }
"const"     { return CONST; }
"const_cast"    { return CONST_CAST; }
"continue"    { return CONTINUE; }
"default"    { return DEFAULT; }
"delete"    { return DELETE; }
"do"     { return DO; }
"double"    { return DOUBLE; }
"dynamic_cast"    { return DYNAMIC_CAST; }
"else"     { return ELSE; }
"enum"     { return ENUM; }
"explicit"    { return EXPLICIT; }
"export"    { return EXPORT; }
"extern"    { return EXTERN; }
"false"     { return FALSE; }
"float"     { return FLOAT; }
"for"     { return FOR; }
```

```
"friend"    { return FRIEND; }
"goto"     { return GOTO; }
"if"     { return IF; }
"inline"    { return INLINE; }
"int"     { return INT; }
"long"     { return LONG; }
"mutable"    { return MUTABLE; }
"namespace"    { return NAMESPACE; }
"new"     { return NEW; }
"operator"    { return OPERATOR; }
"private"    { return PRIVATE; }
"protected"    { return PROTECTED; }
"public"    { return PUBLIC; }
"register"    { return REGISTER; }
"reinterpret_cast"   { return REINTERPRET_CAST; }
"return"    { return RETURN; }
"short"     { return SHORT; }
"signed"    { return SIGNED; }
"sizeof"    { return SIZEOF; }
"static"    { return STATIC; }
"static_cast"    { return STATIC_CAST; }
"struct"    { return STRUCT; }
"switch"    { return SWITCH; }
"template"    { return TEMPLATE; }
"this"     { return THIS; }
"throw"     { return THROW; }
"true"     { return TRUE; }
"try"     { return TRY; }
"typedef"    { return TYPEDEF; }
"typeid"    { return TYPEID; }
"typename"    { return TYPENAME; }
"union"     { return UNION; }
"unsigned"    { return UNSIGNED; }
"using"     { return USING; }
"virtual"    { return VIRTUAL; }
"void"     { return VOID; }
"volatile"    { return VOLATILE; }
"wchar_t"    { return WCHAR_T; }
"while"     { return WHILE; }

[a-zA-Z_][a-zA-Z_0-9]*
{ return check_identifier(yytext); }

"0"[xX][0-9a-fA-F]+{intsuffix}? { return INTEGER; }
"0"[0-7]+{intsuffix}?      { return INTEGER; }
[0-9]+{intsuffix}?  { return INTEGER; }

{fracconst}{exppart}?{floatsuffix}? { return FLOATING; }
[0-9]+{exppart}{floatsuffix}?    { return FLOATING; }

"'"{chartext}*"'" { return CHARACTER; }
"L'"{chartext}*"'" { return CHARACTER; }

"\""{stringtext}*"\""      { return STRING; }
"L\""{stringtext}*"\""     { return STRING; }
.      { fprintf(stderr,
  "%d: unexpected character `%c'\n", lineno,
     yytext[0]); }

%%

static int
yywrap(void)
```

```
{
	return 1;
}
static void
skip_comment(void)
{
	int c1, c2;

	c1 = input();
	c2 = input();

	while(c2 != EOF && !(c1 == '*' && c2 == '/'))
	{
		if (c1 == '\n')
			++lineno;
		c1 = c2;
		c2 = input();
	}
}
static void
skip_until_eol(void)
{
	int c;

	while ((c = input()) != EOF && c != '\n')
		;
	++lineno;
}

static int
check_identifier(const char *s)
{
	/*
	 * This function should check if `s' is a
	 * typedef name or a class
	 * name, or a enum name, ... etc. or
	 * an identifier.
	 */
	switch (s[0]) {
	case 'D': return TYPEDEF_NAME;
	case 'N': return NAMESPACE_NAME;
	case 'C': return CLASS_NAME;
	case 'E': return ENUM_NAME;
	case 'T': return TEMPLATE_NAME;
	}
	return IDENTIFIER;
}
```

Parsing

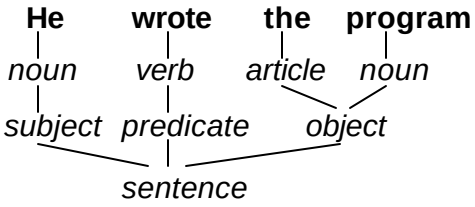
We now move the second module of the front-end: the parser. Recall the front-end components:



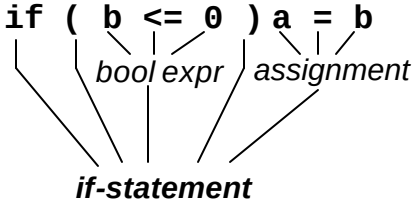
The parser checks the stream of words (tokens) and their parts of speech for grammatical correctness. It determines if the input is *syntactically* well formed. It guides context-sensitive ("semantic") analysis (type checking). Finally, it builds IR for source program.

Syntactic Analysis

Consider the sentence "He wrote the program". The structure of the sentence can be described using grammar syntax of English language.



The analogy can be carried over to syntax of sentences in a programming language. For example, an `if`-statement has the syntax



The parser ensures that sentences of a programming language that make up a program abide by the syntax of the language. If there are errors, the parser will detect them and reports them accordingly. Consider the following code segment that contains a number of syntax errors:

```
int* foo(int i, int j))
{
 for(k=0; i j; )
  fi( i > j )
    return j;
}
```

It is clear that a scanner based upon regular expressions will not be able to detect syntax error.