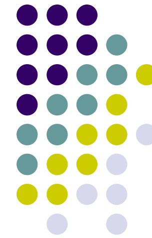
CS 460

Programming Languages
Fall 2008
Dr. Watts
(Week 7)



Project 1

7. Additional specifications may be discussed in class.

- Directory must include a makefile (or Makefile)
- Entering the command ‘make’ must create an executable
- The executable must be called ‘proj1’

Project 1 Submission

- Create a directory called yourlastnameP1
- Copy *only* those files need to compile and execute your project to yourlastnameP1
- With yourlastnameP1 as pwd:

```
chmod 644 *
```

- With parent of yourlastnameP1 as pwd:

```
chmod 700 yourlastnameP1
tar cfvz yourlastnameP1.tgz yourlastnameP1
umask 033 or umask 037
chmod 644 yourlastnameP1.tgz
cp yourlastnameP1.tgz ~tiawatts/cs460drop
umask 077
```

- Check submission page

Project 1 Questions

- Question: In the project description get_lexeme returns a char * and report_error receives a char *. Since I am using C++ can these be strings?
- Answer: Yes
- Question: Should I check to see that an identifier has no more than 32 characters?
- Answer: Don't worry about the 32 character maximum requirement.

Project 1 Questions

- Question: Can function prototypes be different from the specs? Will you be running our functions or will you be running our driver program?
- Answer: I will be running your driver program - so, you can modify the functions as you wish. You will need those functions for part 2 of the project.
- Question: How should sizeof() be handled? Should I use an if statement that checks to see if you have sizeof and then allows () to be valid?
- Answer: Don't worry about the parenthesis after sizeof - just generate a SIZEOF token when you find an identifier that is "sizeof". This should require a single if statement.

Project 1 Questions

- Question: When should I decrement the position counter?
- Answer: You should decrement the position counter any time you need to see a character that is not part of the lexeme to determine that you have reached the end of the lexeme. For example, if you see a '51+abc' in your input, you have to see the '+' to recognize that you have reached the end of the numeric literal (51) and you have to see the 'a' to recognize that the '+' is a single character symbol '+' and not the first part of '++' or '+=‘; so, in these cases you will need to decrement the position counter. If you have identified a '++' or '+=', you probably do not need to decrement the position counter.

Project 1 Questions

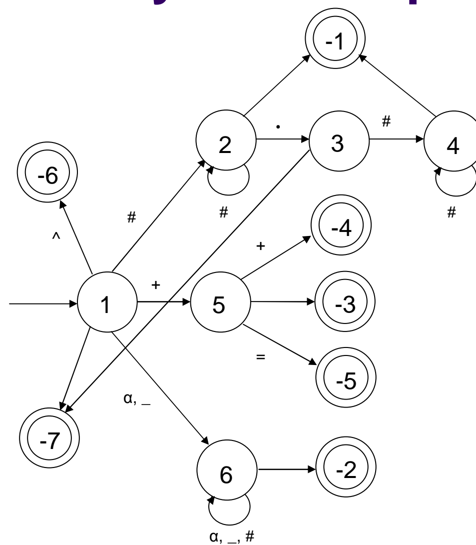
- Question: What is the purpose of an enumerated type?
- Answer: The enumerated type helps with readability. For example,

```
switch (token)
{
      case PLUS: etc.
      case MINUS: etc;
}
```

is much clearer than

```
switch (token)
{
      case 3: etc;
      case 4: etc;
}
```

Lexical Analysis Example 2



Lexical Analysis Example 2

state	a	1	_	+	=	^	.	other
1	6	2	6	5	-7*	-6	-7	-7
2	-1	2	-1	-1	-1	-1	3	-1
3	-7	4	-7	-7	-7	-7	-7	-7
4	-1	4	-1	-1	-1	-1	-1	-1
5	-3	-3	-3	-4	-5	-3	-3	-3
6	6	6	6	-2	-2	-2	-2	-2

```
enum token_type {NUMLIT = 1, IDENT, PLUS, PLUSPLUS, PLUSEQ,
                 XOR, ERROR, END};
```

Lexical Analysis Example 2

- **makefile**

```
ex2 : lex.o main.o
        g++ -g -o ex2 lex.o main.o
lex.o : lex.h lex.cpp
        g++ -g -c lex.cpp
main.o : lex.h main.cpp
        g++ -g -c main.cpp
clean :
        rm ex2 *.o
```

Lexical Analysis Example 2

- **lex.h**

```
#ifndef LEX_H
#define LEX_H
#include <iostream>
#include <iomanip>
#include <fstream>
#include <string>
#include <cstdlib>
using namespace std;
enum token_type {NUMLIT = 1, IDENT, PLUS, PLUSPLUS, PLUSEQ,
   XOR, ERROR, END};
void open_file (char * filename);
void close_file ();
string get_lexeme ();
token_type get_token ();
#endif
```

Lexical Analysis Example 2

- **main.cpp**

```
#include "lex.h"
int main (int argc, char * argv[])
{
     open_file (argv[1]);
     token_type token;
     do
     {
          token = get_token ();
     } while (token != END);
     close_file();
     return 0;
}
```

Lexical Analysis Example 2

- **lex.cpp**

```
#include "lex.h"
ifstream input;
string token_names[] = {"", "NUMLIT", "IDENT", "PLUS", "PLUSPLUS",
   "PLUSEQ", "XOR", "ERROR", "END"};
string lexeme;
void open_file (char * filename)
{
    input.open (filename);
    if (input.fail ())
    {
        cerr << "File " << filename << " not found\n";
        exit (1);
    }
}
void close_file ()
{
    input.close ();
}
string get_lexeme ()
{
    return lexeme;
}
```

Lexical Analysis Example 2

- lex.cpp (continued)

```
token_type get_token ()
{
    char alphabet[] = "a1_+=^.";
    int table[][14] = {{0,0,0,0,0,0,0,0},
             {6,2,6,5,-7,-6,-7,-7}, {-1,2,-1,-1,-1,-1,3,-1},
             {-7,4,-7,-7,-7,-7,-7,-7}, {-1,4,-1,-1,-1,-1,-1,-1},
             {-3,-3,-3,-4,-5,-3,-3,-3}, {6,6,6,-2,-2,-2,-2,-2,-2}};
    static string text = " ";
    static int pos = 0;
    while (isspace (text[pos]))
    {
        pos++;
        if (pos >= text.length())
        {
            getline (input,text);
            if (input.fail())
                return END;
            cout << text << endl;
            text += " ";
            pos = 0;
        }
    }
```

Lexical Analysis Example 2

- lex.cpp (continued)

```
        int start = pos;
        int state = 1;
        while (state > 0)
        {
            int a;
            char temp = text[pos];
            if (isalpha (temp)) temp = 'a';
            else if (isdigit (temp)) temp = '1';
            for (a = 0; alphabet[a] && alphabet[a] != temp; a++);
            state = table [state][a];
            pos++;
        }
        token_type token = token_type(-state);
        if (token < PLUSPLUS)
            pos--;
        int end = pos;
        lexeme = text.substr (start, end - start);
        cout << '\t' << left << setw (12) << token_names [token] <<
                       lexeme << endl;
        return token;
    }
```

What is Parsing?

- Process of analyzing syntax
- Determining if the order of the tokens generated for the lexemes of the input are in a legal order according to some grammar
- Creation of a parse tree
  - Explicit or implicit
- Error recovery
  - When an error is detected, the parser must get back to a normal state and continue analysis of the input
- Basis for translation

Parsing

- Top Down
  - Recursive Descent
  - LL Parsing – Left to right scan of the input; Leftmost derivation
- Bottom Up
  - Shift Reduce
  - LR Parsing – Left to right scan of the input; Rightmost derivation

Recursive Descent Parser

- Collection of subprograms
- May be recursive
- Produces a parse tree in top-down (descending) order
  - Implicit tree traced
- Works well with LL grammar defined by BNF (or EBNF)
- Right recursion OK – Left recursion causes problems

<A> → c <A>	vs	<A> → <A> c
<A> → λ		<A> → λ

LL(1) Grammar for a Small Programming Language

T = {begin, end, ;, =, A, B, C, +, *}
N = {<program>, <stmt_list>, <stmt>, <var>, <stmt_tail>, <expression>, <expr_tail>}
Start = <program>

P =
{
1. <program> → **begin** <stmt_list> **end**
2. <stmt_list> → <stmt> <stmt_tail>
3. <stmt_tail> →**;** <stmt_list>
4. <stmt_tail> → λ
5. <stmt> → <var> **=** <expression>
6. <var> → **A**
7. <var> → **B**
8. <var> → **C**
9. <expression> → <var><expr_tail>
10. <expr_tail> → + <var><expr_tail>
11. <expr_tail> → * <var><expr_tail>
12. <expr_tail> → λ

}

Example 1:
```
begin
    A = B + C;
    C = A * B
end
```

Example 2:
```
begin
    A = B + A * C;
    C = A * B;
end
```

Derivation of Example 1

Start	<program>
1	**begin** <stmt_list> end
2	<stmt> <stmt_tail> end
5	<var> = <expression> <stmt_tail> end
6	**A =** <expression> <stmt_tail> end
9	<var> <expr_tail> <stmt_tail> end
7	**B** <expr_tail> <stmt_tail> end
10	**+** <var> <expr_tail> <stmt_tail> end
8	**C** <expr_tail> <stmt_tail> end
12	λ <stmt_tail> end
3	**;** <stmt_list> end
2	<stmt> <stmt_tail> end
5	<var> = <expression> <stmt_tail> end
8	**C =** <expression> <stmt_tail> end
9	<var> <expr_tail> <stmt_tail> end
6	**A** <expr_tail> <stmt_tail> end
11	* <var> <expr_tail> <stmt_tail> end
7	**B** <expr_tail> <stmt_tail> end
12	λ <stmt_tail> end
4	λ **end**

- Done

Parsing of Example 1

From main parsing routine, call <program> function
From <program>, match **begin**; call <stmt_list> function
From <stmt_list>, see **A** call <stmt> function
From <stmt>, see **A** call <var> function
From <var>, match **A**; return
From <stmt>, match =; see **B** call <expression> function
From <expression>, see **B** call <var> function
From <var>, match **B**; return
From <expression>, see **+** call <expr_tail> function
From <expr_tail>, match **+**; see **C** call <var> function
From <var>, match **C**; return
From <expr_tail>, call <expr_tail>
From <expr_tail>, see **;** return
From <expr_tail>, return
From <expression>, return
…
From <program>, match end; return
From main parsing routine, print error count and terminate

Recursive Descent Parser for a Small Programming Language

1) <program> → **begin** <stmt_list> **end**

```
program ()
{
   if (current_token == begin)
   { // rule 1
        get_next_token;
        call stmt_list;
        if (current_token == end)
                get_next_token;
        else
                call error_routine;
   }
   else
        call error_routine;
   return;
}
```

Recursive Descent Parser for a Small Programming Language

2) <stmt_list> → <stmt> <stmt_tail>

```
stmt_list ()
{ // rule 2
     call stmt;
     call stmt_tail;
     return;
}
```

Recursive Descent Parser for a Small Programming Language

3) <stmt_tail> →; <stmt_list>
4) <stmt_tail> → λ

```
stmt_tail ()
{
   if (current_token == ;)
   { // rule 3
       get_next_token;
       call stmt_list;
   }
   else if (current_token == end)
   { // rule 4
   }
   else
       call error_routine;
   return;
}
```

Recursive Descent Parser for a Small Programming Language

5) <stmt> → <var> **=** <expression>

```
stmt ()
{ // rule 5
   call var;
   if (current_token == =)
   {
       get_next_token;
       call expression;
   }
   else
       call error_routine;
   return;
}
```

Recursive Descent Parser for a Small Programming Language

6) <var> → **A**
7) <var> → **B**
8) <var> → **C**

```
var ()
{ // rule 5
   if (current_token == A)
   { // rule 6
       get_next_token;
   }
   else if (current_token == B)
   { // rule 7
       get_next_token;
   }
   else if (current_token == C)
   { // rule 8
       get_next_token;
   }
   else
       call error_routine;
   return;
}
```

Recursive Descent Parser for a Small Programming Language

9) <expression> → <var><expr_tail>

```
expression ()
{ // rule 9
     call var;
     call expr_tail;
     return;
}
```

Recursive Descent Parser for a Small Programming Language

10) <expr_tail> → + <var><expr_tail>
11) <expr_tail> → * <var><expr_tail>
12) <expr_tail> → λ

```
expr_tail ()
{
    if (current_token == +)
    { // rule 10
          get_next_token;
          call var;
          call expr_tail;
    }
    else if (current_token == *)
    { // rule 11
          get_next_token;
          call var;
          call expr_tail;
    }
    else if (current_token == ; or current_token == end)
    { // rule 12
    }
    else
          call error_routine;
    return;
}
```

Context Free Grammar Definition

- Given a Context Free Grammar of the form:
  - Terminals = $\{T_1, T_2, T_3, \ldots\}$
  - Non-terminals = $\{\langle nt_1\rangle, \langle nt_2\rangle, \langle nt_3\rangle, \ldots\}$
  - A Start symbol from the set of non-terminals
  - A set of Production rules of the form

    $\langle nt_i\rangle \rightarrow$ string of T and <nt> symbols

First and Follow Sets

- Firsts
  - A terminal symbol T_i is a member of the First Set of non-terminal symbol $\langle nt_j\rangle$ if T_i can become the first terminal symbol in a complete expansion of $\langle nt_j\rangle$.
- Follows
  - A terminal symbol T_i is a member of the Follow Set of non-terminal symbol $\langle nt_j\rangle$ if T_i can become the first terminal symbol immediately following a complete expansion of $\langle nt_j\rangle$.