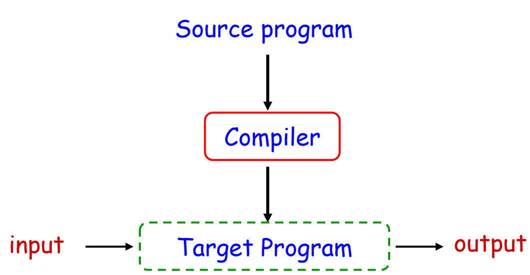
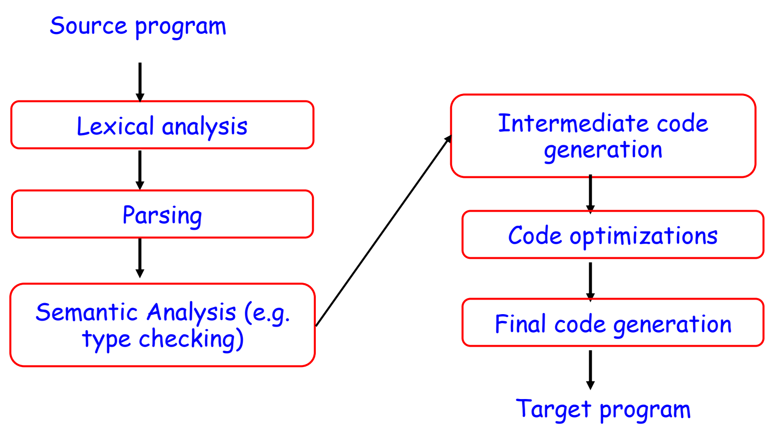
CS571: Programming Languages

Compiler

- Translates source code into target code. The user may execute the target code.
  - * The source and the target programs must be semantically equivalent – the compilation process must be meaning preserving.

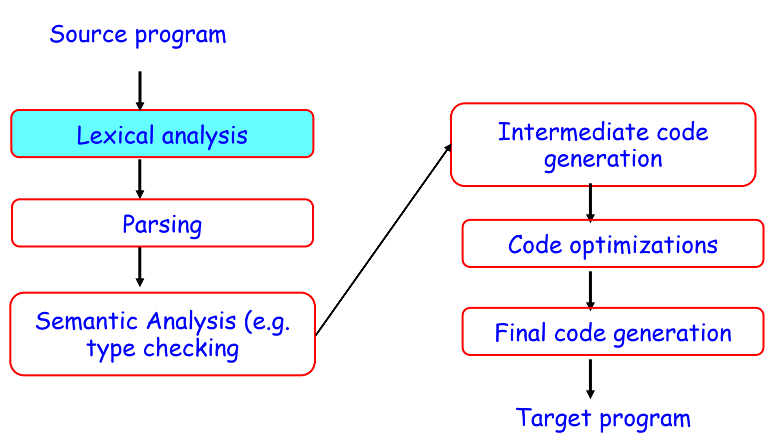


Phase of Compilation



3

Phase of Compilation



4

Lexical Analysis (Scanning or Tokenizing)

- Identify the words: converts a stream of characters into a stream of tokens.
  - Token: name given to a family of words
    - Keywords: e.g. if and while
    - Literals or constants: e.g. 12, “hello”
    - Special symbols: e.g. “;”, “<=“
    - Identifiers: e.g. x, y, z
  - Is the following legal in C: int if;
  - Is the following legal in C: int whileif;

CS571 Programming Languages

5

5

Lexical Analysis (Scanning or Tokenizing)

- Identify the words: converts a stream of characters into a stream of tokens.
  - Token: name given to a family of words
    - Keywords: e.g. if and while
    - Literals or constants: e.g. 12, “hello”
    - Special symbols: e.g. “;”, “<=“
    - Identifiers: e.g. x, y, z
  - Is the following legal in C: int if;
  - Is the following legal in C: int whileif;

Principle of Longest Substring

6

6

Lexical Analysis: Example

foo = 1 - 3**2

Lexeme	Token Type
foo	Variable
=	Assignment Operator
1	Number
-	Subtraction Operator
3	Number
**	Power Operator
2	Number

Lexical Analysis (Scanning or Tokenizing)

- Lexical analyzer discards white space, tabs and comments between the tokens.
- The format of program can affect the way tokens are recognized.
- How do we compactly represent the set of all strings corresponding to a token?

Lexical Analysis (Scanning or Tokenizing)

- Lexical analyzer discards white space, tabs and comments between the tokens.
- The format of program can affect the way tokens are recognized.
- How do we compactly represent the set of all strings corresponding to a token?
  - E.g. **integer** represents the set of all integers, i.e. all sequences of digits (0-9), preceded by an optional sign (+ or -)
  - Can we simply enumerate all integers?
- Solution: use regular expression

9

9

Regular Expressions

- Three basic operations: concatenation, repetition (*) and choice (|).
  - ∍: empty string
  - a: the set {a} that contains a single string a
  - ab: the set {ab} that contains a single string ab
  - a*: the set {∍, a, aa, …} that contains all strings of zero or more a's.
  - a|b: the set {a,b} that contains two strings a and b
    - Analogous to union

10

10

Regular Expression: Example

- (a|b)*:

Regular Expression: Example

- (a|b)*: set of strings with zero or more a's and zero or more b': {∍, a, b, aa, ab, ba, bb, aaa, aab, …}
- (a*b*):

Regular Expression: Example

- (a|b)*: set of strings with zero or more a's and zero or more b': {∍, a, b, aa, ab, ba, bb, aaa, aab, …}
- (a*b*): set of strings with zero or more a's and zero or more b's such that all a's occur before any b: {∍, a, b, aa, ab, bb, aaa, aab, …}
- (a|b)(a|b):

Regular Expression: Example

- (a|b)*: set of strings with zero or more a's and zero or more b': {∍, a, b, aa, ab, ba, bb, aaa, aab, …}
- (a*b*): set of strings with zero or more a's and zero or more b's such that all a's occur before any b: {∍, a, b, aa, ab, bb, aaa, aab, …}
- (a|b)(a|b): set {aa, ab, ba, bb}

Regular Expressions (Cont.)

- Additional operations
  - [-]: range of characters
  - +: one or more repetitions
  - ?: option
- Example
  - [0-9]+:
  - [+|-]?[0-9]+:

Regular Expressions (Cont.)

- Additional operations
  - [-]: range of characters
  - +: one or more repetitions
  - ?: option
- Example
  - [0-9]+:

    1 or more digits (characters between 0 and 9)
  - [+|-]?[0-9]+:

Regular Expressions (Cont.)

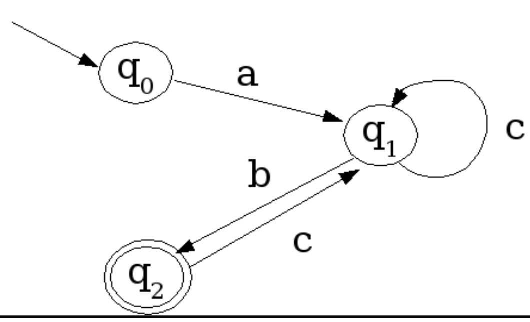
- Additional operations
  - [-]: range of characters
  - +: one or more repetitions
  - ?: option
- Example
  - [0-9]+:
    1 or more digits (characters between 0 and 9)
  - [+|-]?[0-9]+:
    positive/negative integers

17

17

Lexical Analysis

- Regular expressions are used to specify the set of strings corresponding to a token.
- An automaton (DFA/NFA) is built from the above specifications.
- Each final state is associated with an action: e.g. accept the token.

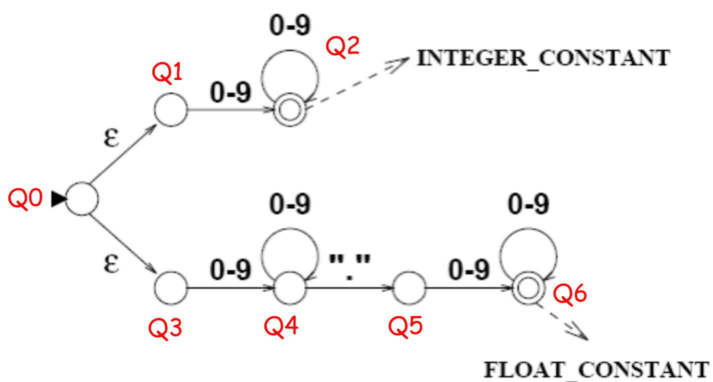


18

18

Specifying Lexical Analysis

- [0-9]+ {return(INTEGER_CONSTANT)}
- [0-9]+"."[0-9]+ {return(FLOAT_CONSTANT)}



19

19

- Draw an automaton that accepts the regular expression a(b|c)d+

20

20

FLEX (Fast LEXical analyzer generator)

- Tool for generating lexical analyzers
- Input: lexical specifications (.l file)
- Output: A C source file named "lex.yy.c" that defines the function yylex(), which returns a token on each invocation

.l file → FLEX → Lex.yy.c

lex.yy.c → C compiler → a.out

21

21

FLEX (Fast LEXical analyzer generator)

- When the analyzer executes, it analyzes input, looking for strings that match any of its patterns.
- Once the match is determined
  - the corresponding token is stored in the global character pointer/array yytext
  - The length of the token is stored in the global integer yyleng.

input → a.out → sequence of tokens

22

22

FLex Specifications

%{

C headers, C code

%}

Regular definitions e.g.:

digit [0-9] //integers from 0 through 9

%%

Token Specifications e.g.:

{digit}+ // Use the previously specified regular
// definition digit.

%%

main function

23

23

Example I

```
%{
#include <stdio.h>
%}
digit           [0-9]
%%
"+"             {printf("plus");}
"-"             {printf("minus");}
{digit}+        {printf("integer");}
.               {printf("syntax error");}
%%
int main(void){
      yylex();
      return 0;
}
```


24

24

Example II (example.l)

```
%{
int num_lines = 0, num_chars = 0;
%}

%%
\n      {++num_lines; ++num_chars;}
.       {++num_chars;}
%%

main(){
  yylex();
  printf("# of lines = %d, # of chars = %d\n", num_lines, num_chars);
}
```


Compiling example.l

```
bingsun2% flex example.l
bingsun2% gcc -o demo lex.yy.c -lfl
bingsun2% ./demo
adfs
bafdfd
Cadsg
Ctrl-D
# of lines = 3, # of chars = 18
bingsun2%
```

Example

- Write a flex program that prints all integers in a file. The file can contain any symbol.

E.g. file **f**
1dfe45fgk6

The program outputs:

1
45
6

27

27

```
%{
#include <stdio.h>
%}

digit       [0-9]

%%
{digit}+  {printf("%d\n", atoi(yytext));}
.      { }

%%
int main(void){ yylex(); return 0; }
```

28

Example

- Write a flex program that searches for the string **"abc"** in file **f**, and prints line numbers and locations of string "abc". The file can contain any symbol.

E.g. file **f**
helloabcworldabc
helloworld
abchelloword

The program outputs:

Line number: 1, location: 6
Line number: 1, location: 14
Line number: 3, location: 1

29

29

```
%{
int num_lines = 1, num_chars = 1;
%}

%%
"abc"   {printf("Line number: %d, ", num_lines);
         printf("Location: %d\n", num_chars);
         num_chars=num_chars+3;}
\n      {++num_lines; num_chars=1;}
.       {++num_chars;}
%%

main(){ yylex();}
```

30

Lexical Analysis: A Summary

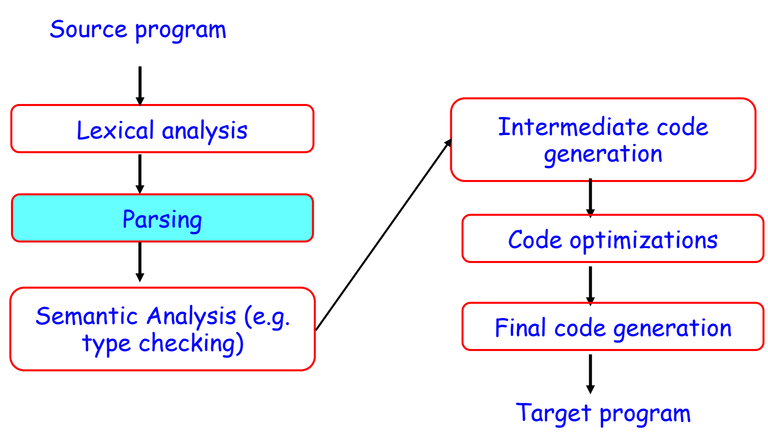
- Convert a steam of characters into a stream of tokens
  - Detect errors such as misspelling an identifier, keyword, or operator.
  - Strip off comments
  - Recognize line numbers
  - Ignore white space characters

- FLEX tutorial:
- ftp://ftp.gnu.org/old-gnu/Manuals/flex-2.5.4/html_mono/flex.html

31

31

Phase of Compilation



32

32

Parsing

- Syntax analysis
- Obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by the grammar for the source language.
- Detect syntax errors, such as arithmetic expression with unbalanced parentheses.

33

33

Requirements for Parser

- Should report the presence of errors clearly and accurately
- Should recover from each error quickly enough to be able to detect subsequent errors.
- Should not significantly slow down the processing of correct programs.

CS571 Programming Languages

34

34

Structure of a Language

- Grammars: notation to succinctly represent the structure of a language
- Programming languages tend to be specified in terms of a context-free grammar

$$E \rightarrow \quad E + E \mid E - E \mid E * E \mid E / E \mid (E) \mid Number$$
$$Number \rightarrow \quad Number\ Digit \mid Digit$$
$$Digit \rightarrow \quad 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

$\boxed{\text{E} \rightarrow \text{E+ E}}$ product

35

35

Context-Free Grammars

- Nonterminals: can be broken down into further structure
  - Specified using capital letters.

Nonterminal

E –> E+E | …

- Terminals
  - Specified using lower-case letters.

Terminals

Digit –> 0 | 1 | …

36

36

Context-Free Grammars

- A distinguished start symbol.

Start symbol

$\boxed{E} \rightarrow E+E|\ldots|Number$
$Number \rightarrow Number\ Digit\ |\ Digit$
$Digit \rightarrow 0\ |\ 1\ |\ \ldots$

37

37

Derivation

- $\alpha \rightarrow \beta$: β is derivable from α in one step.
  - α, β: 0 or more terminals or nonterminals
- $\alpha \rightarrow \beta$ if
  - $\alpha = \alpha_1 A \alpha_2$
  - $\beta = \alpha_1 \gamma \alpha_2$
  - $A \rightarrow \gamma$ is a product in the context-free grammar

$S \rightarrow \varepsilon$
$S \rightarrow 0S1$

$000S111 \rightarrow 0000S1111$

- We write $\alpha \rightarrow^* \beta$ if β is derivable from α in multiple steps.

$\alpha \rightarrow^* \beta$ if $\alpha = \alpha_0 \rightarrow \alpha_1 \rightarrow \ldots \rightarrow \alpha_{n-1} \rightarrow \alpha_n = \beta$

38

38

Derivation

- $\alpha \rightarrow \beta$: β is derivable from α in one step.
  - α, β: 0 or more terminals or nonterminals
- $\alpha \rightarrow \beta$ if
  - $\alpha = \alpha_1 A \alpha_2$
  - $\beta = \alpha_1 \gamma \alpha_2$
  - $A \rightarrow \gamma$ is a product in the context-free grammar

$S \rightarrow \varepsilon$
$S \rightarrow 0S1$

$$\underbrace{\overbrace{000}^{\alpha_1}\overbrace{\boxed{S}}^{A}\overbrace{111}^{\alpha_2}}_{\alpha} \rightarrow \underbrace{\overbrace{000}^{\alpha_1}\overbrace{\boxed{0S1}}^{\gamma}\overbrace{111}^{\alpha_2}}_{\beta}$$

- We write $\alpha \rightarrow^* \beta$ if β is derivable from α in multiple steps.

$\alpha \rightarrow^* \beta$ if $\alpha = \alpha_0 \rightarrow \alpha_1 \rightarrow \ldots \rightarrow \alpha_{n-1} \rightarrow \alpha_n = \beta$

39

39

Parse Tree

- A graphical representation for a derivation
- Arithmetic expressions with operators + and *.

$E \rightarrow E+E \mid E*E \mid (E) \mid Digit$
$Digit \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

The parse tree of *(3+4) * 5* is

40

40

Parse Tree

- A graphical representation for a derivation
- Arithmetic expressions with operators + and *.

$E \rightarrow E{+}E \mid E{*}E \mid (E) \mid Digit$
$Digit \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

The parse tree of $(3+4) * 5$ is

E

41

41

Parse Tree

- A graphical representation for a derivation
- Arithmetic expressions with operators + and *.

$E \rightarrow E{+}E \mid E{*}E \mid (E) \mid Digit$
$Digit \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

The parse tree of $(3+4) * 5$ is

E

E * E

42

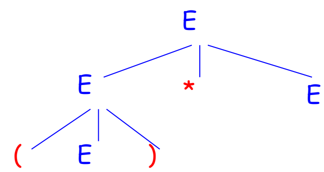
42

Parse Tree

- A graphical representation for a derivation
- Arithmetic expressions with operators + and *.

$E \rightarrow E+E \mid E*E \mid (E) \mid Digit$
$Digit \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

The parse tree of $(3+4) * 5$ is



43

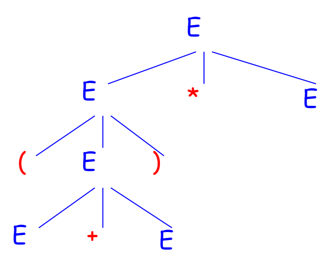
43

Parse Tree

- A graphical representation for a derivation
- Arithmetic expressions with operators + and *.

$E \rightarrow E+E \mid E*E \mid (E) \mid Digit$
$Digit \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

The parse tree of $(3+4) * 5$ is



44

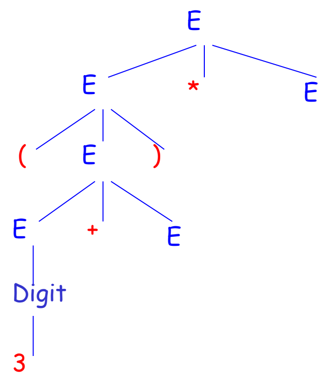
44

Parse Tree

- A graphical representation for a derivation
- Arithmetic expressions with operators + and *.

$E \rightarrow E+E \mid E*E \mid (E) \mid Digit$
$Digit \rightarrow \; 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

The parse tree of $(3+4) * 5$ is



45

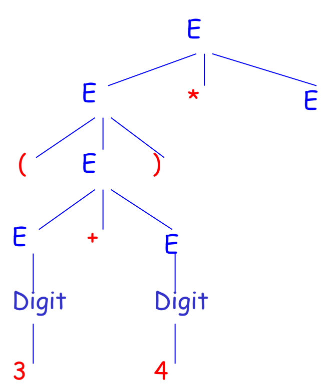
45

Parse Tree

- A graphical representation for a derivation
- Arithmetic expressions with operators + and *.

$E \rightarrow E+E \mid E*E \mid (E) \mid Digit$
$Digit \rightarrow \; 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

The parse tree of $(3+4) * 5$ is



46

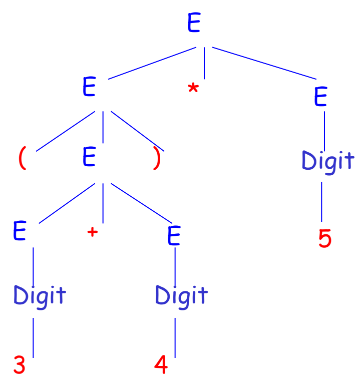
46

Parse Tree

- A graphical representation for a derivation
- Arithmetic expressions with operators + and *.

*E → E+E | E*E | (E) | Digit*
Digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

The parse tree of *(3+4) * 5* is



Parse Tree: Ambiguity

- Arithmetic expressions with operators + and *.

*E → E+E | E*E | (E) | Digit*
Digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

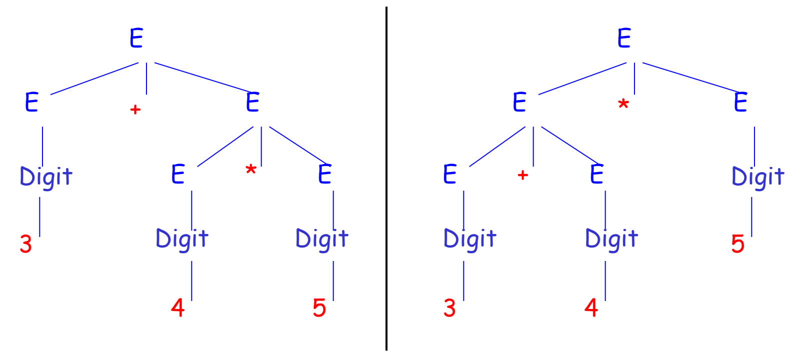
The parse tree of *3+4*5* is

Parse Tree: Ambiguity

- Arithmetic expressions with operators + and *.

$$E \rightarrow E{+}E \mid E{*}E \mid (E) \mid Digit$$
$$Digit \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

The parse tree of $3+4*5$ is



Parsers

- Bottom-up (shift-reduce): construct the parse trees from the leaves to the root
  - E.g. YACC and Bison
- Top-down: nonterminals are expanded to match incoming tokens and directly construct a derivation.
  - E.g. LL parser.
- Recursive-descent parsing: turning the nonterminals into a group of mutually recursive procedures whose actions are based on the right-hand sides of CFG.

Recursive Descent Parsing

- Turning the nonterminals into a group of mutually recursive procedures whose actions are based on the rhs of CFG.
- Example:

  Stmt → If_stmt | While_stmt | …
  If_stmt → if (E) Stmt | …

Recursive-descent code:

```
void ifStmt()
{  match("if");
   match("(");
   expression();
   match(")");
   statement();
}
```

51

51

Bottom-Up (Shift-Reduce) Parsers

- Construct the parse trees from the leaves to the root
- Stack implementation
  - Stack: Grammar symbols. $: end of the stack
  - Input buffer: String w to be parsed. $: end of the input
  - Actions:
    - Shift: the next input symbol is shifted onto the top of the stack.
    - Reduce: reduce the strings in the stack to the left-side of the corresponding CFG.
  - The parser terminates if it has detected an error or (the stack contains the start symbol and the input is empty)

52

52

Bottom-Up (Shift-Reduce) Parsers

- Example: $E \rightarrow E+E \mid E*E \mid (E) \mid Id$
  $Id \rightarrow id1 \mid id2 \mid id3$
- Input: *id1+id2*id3 (assume that * has higher precedence than +)*

STACK	INPUT	ACTION
(1) $	id1+ id2 * id3$	Shift
(2) $id1	+ id2 * id3$	Reduce by Id -> id1, E → Id
(3) $E	+ id2 * id3$	

53

53

Bottom-Up (Shift-Reduce) Parsers

- Example: $E \rightarrow E+E \mid E*E \mid (E) \mid Id$
  $Id \rightarrow id1 \mid id2 \mid id3$
- Input: *id1+id2*id3 (assume that * has higher precedence than +)*

STACK	INPUT	ACTION
(1) $	id1+ id2 * id3$	Shift
(2) $id1	+ id2 * id3$	Reduce by Id -> id1, E → Id
(3) $E	+ id2 * id3$	Shift
(4) $E +	id2 * id3$	

54

54

Bottom-Up (Shift-Reduce) Parsers

- Example: *E → E+E | E*E | (E) | Id*
  Id → id1 | id2 | id3
- Input: *id1+id2*id3 (assume that * has higher precedence than +)*

STACK	INPUT	ACTION
(1) $	id1+ id2 * id3$	Shift
(2) $id1	+ id2 * id3$	Reduce by Id -> id1, E → Id
(3) $E	+ id2 * id3$	Shift
(4) $E +	id2 * id3$	Shift
(5) $E + id2	* id3$	

55

55

Bottom-Up (Shift-Reduce) Parsers

- Example: *E → E+E | E*E | (E) | Id*
  Id → id1 | id2 | id3
- Input: *id1+id2*id3 (assume that * has higher precedence than +)*

STACK	INPUT	ACTION
(1) $	id1+ id2 * id3$	Shift
(2) $id1	+ id2 * id3$	Reduce by Id -> id1, E → Id
(3) $E	+ id2 * id3$	Shift
(4) $E +	id2 * id3$	Shift
(5) $E + id2	* id3$	Reduce by Id -> id2, E → Id
(6) $E + E	* id3$	

56

56

Bottom-Up (Shift-Reduce) Parsers

- Example: $E \rightarrow E+E \mid E*E \mid (E) \mid Id$
  $Id \rightarrow id1 \mid id2 \mid id3$
- Input: *id1+id2*id3 (assume that * has higher precedence than +)*

STACK	INPUT	ACTION
(1) $	id1+ id2 * id3$	Shift
(2) $id1	+ id2 * id3$	Reduce by Id -> id1, E → Id
(3) $E	+ id2 * id3$	Shift
(4) $E +	id2 * id3$	Shift
(5) $E + id2	* id3$	Reduce by Id -> id2, E → Id
(6) $E + E	* id3$	Shift
(7) $E + E *	id3$	

57

57

Bottom-Up (Shift-Reduce) Parsers

- Example: $E \rightarrow E+E \mid E*E \mid (E) \mid Id$
  $Id \rightarrow id1 \mid id2 \mid id3$
- Input: *id1+id2*id3 (assume that * has higher precedence than +)*

STACK	INPUT	ACTION
(1) $	id1+ id2 * id3$	Shift
(2) $id1	+ id2 * id3$	Reduce by Id -> id1, E → Id
(3) $E	+ id2 * id3$	Shift
(4) $E +	id2 * id3$	Shift
(5) $E + id2	* id3$	Reduce by Id -> id2, E → Id
(6) $E + E	* id3$	Shift
(7) $E + E *	id3$	Shift
(8) $E + E * id3	$	

58

58

Bottom-Up (Shift-Reduce) Parsers

- Example: *E → E+E | E*E | (E) | Id*
  Id → id1 | id2 | id3
- Input: *id1+id2*id3 (assume that * has higher precedence than +)*

STACK	INPUT	ACTION
(1) $	id1+ id2 * id3$	Shift
(2) $id1	+ id2 * id3$	Reduce by Id -> id1, E → Id
(3) $E	+ id2 * id3$	Shift
(4) $E +	id2 * id3$	Shift
(5) $E + id2	* id3$	Reduce by Id -> id2, E → Id
(6) $E + E	* id3$	Shift
(7) $E + E *	id3$	Shift
(8) $E + E * id3	$	Reduce by Id -> id3, E → Id
(9) $E + E * E	$	

59

59

Bottom-Up (Shift-Reduce) Parsers

- Example: *E → E+E | E*E | (E) | Id*
  Id → id1 | id2 | id3
- Input: *id1+id2*id3 (assume that * has higher precedence than +)*

STACK	INPUT	ACTION
(1) $	id1+ id2 * id3$	Shift
(2) $id1	+ id2 * id3$	Reduce by Id -> id1, E → Id
(3) $E	+ id2 * id3$	Shift
(4) $E +	id2 * id3$	Shift
(5) $E + id2	* id3$	Reduce by Id -> id2, E → Id
(6) $E + E	* id3$	Shift
(7) $E + E *	id3$	Shift
(8) $E + E * id3	$	Reduce by Id -> id3, E → Id
(9) $E + E * E	$	Reduce by E → E * E
(10) $E + E	$	

60

60

Bottom-Up (Shift-Reduce) Parsers

- Example: *E → E+E | E*E | (E) | Id*
  Id → id1 | id2 | id3
- Input: *id1+id2*id3 (assume that * has higher precedence than +)*

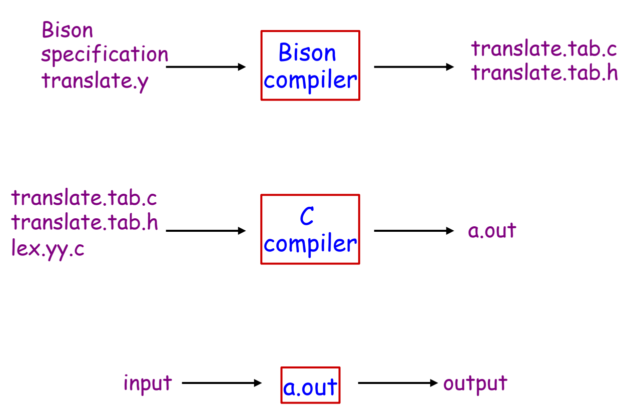
STACK	INPUT	ACTION
(1) $	id1+ id2 * id3$	Shift
(2) $id1	+ id2 * id3$	Reduce by Id -> id1, E → Id
(3) $E	+ id2 * id3$	Shift
(4) $E +	id2 * id3$	Shift
(5) $E + id2	* id3$	Reduce by Id -> id2, E → Id
(6) $E + E	* id3$	Shift
(7) $E + E *	id3$	Shift
(8) $E + E * id3	$	Reduce by Id -> id3, E → Id
(9) $E + E * E	$	Reduce by E → E * E
(10) $E + E	$	Reduce by E → E + E
(11) $E	$	Accept

61

61

Bison

- A general-purpose parser generator



62

62

Bison (Cont.)

- translate.tab.c defines a function yyparse() which implements grammar.
  - Additional functions
    - The lexical analyzer.
    - An error-reporting function yyerror() which the parser calls to report an error.
    - A function called main

Bison (Cont.)

- Four parts

```
%{
C declarations
%}
Bison declarations
%%
Translation rules
%%
C functions
```

Example: Prefix Calculator

- Write a calculator
  - Reads an arithmetic expression
  - Evaluates the expression
  - Prints its numeric value using println

Es → *E;* | *println E;*

E → *E + E* | *E-E* | *E*E* | *E/E* | *integer*

65

65

Flex Code: Prefix Calculator (calc.l)

```
%{
#include <stdio.h>
#include "calc.tab.h"
%}
digit   [0-9]
%%
"println"       { return(TOK_PRINTLN);}
{digit}+        { sscanf(yytext, "%d", &(yylval.int_val));
                  return TOK_NUM; }
";"             { return(TOK_SEMICOLON); }
"+"             { return(TOK_ADD); }
"-"             { return(TOK_SUB); }
"*"             { return(TOK_MUL); }
"/"             { return(TOK_DIV); }
\n              {}
.               {printf("Invalid character '%c'\n", yytext[0]);}
%%
```

66

66

Bison Code: Prefix Calculator (calc.y)

```
%{
#include <stdio.h>
%}
%token TOK_SEMICOLON TOK_ADD TOK_SUB TOK_MUL TOK_DIV
    TOK_NUM TOK_PRINTLN

/*all possible types*/
%union{
      int int_val;
}

%type <int_val> expr TOK_NUM

/*left associative*/
%left TOK_ADD TOK_SUB
%left TOK_MUL TOK_DIV
%%
```

Can we change the order of these two lines. Why?

67

67

Bison Code: Prefix Calculator

```
%{
#include <stdio.h>
%}
%token TOK_SEMICOLON TOK_ADD TOK_SUB TOK_MUL TOK_DIV
    TOK_NUM TOK_PRINTLN

/*all possible types*/
%union{
      int int_val;
}

%type <int_val> expr TOK_NUM

/*left associative*/
%left TOK_ADD TOK_SUB
%left TOK_MUL TOK_DIV
%%
```

Can we change the order of these two lines.
No. change precedence

68

68

Bison Code: Desk Calculator (Cont.)

```
expr_stmt: expr TOK_SEMICOLON
          | TOK_PRINTLN expr TOK_SEMICOLON
            { fprintf(stdout, "the value is %d\n", $2); }
;
expr:
     expr TOK_ADD expr
      { $$ = $1 + $3;}
     | expr TOK_SUB expr
      { $$ = $1 - $3; }
     | expr TOK_MUL expr
      { $$ = $1 * $3;   }
     ……
```


Bison Example: Desk Calculator (Cont.)

```
%%
int yyerror(char *s){
    fprintf(stderr, “syntax error");
    return 0; }

int main()
{
  yyparse();
  return 0;
}
```

Compiling calc.l and calc.y

- flex calc.l //compile calc.l
- bison -dv calc.y //compile calc.y
- gcc -o calc calc.tab.c lex.yy.c -lfl

Compilation

```
bingsun2% ./calc
1+2;
println 2;
the value is 2
Println 1+2*3;
the value is 7
1++2;
syntax error
```

- Bison manual: http://dinosaur.compilertools.net/bison/index.html