

Recursive Descent Parsers

- Read and *recognize* the input (in order to translate it or evaluate it)
- Implicitly construct the derivation tree
- Design is driven by the CFG of the language they recognize

Processing Expressions: A First Example

- Using the CFG for Boolean expressions, let's determine the value of expressions such as (T AND NOT (T))
- Assume expressions are represented as a string of characters with no syntax errors

Rewrite Rules for Boolean Expressions

$\langle \text{bool exp} \rangle \rightarrow F \mid$
 $T \mid$
 $\text{NOT} (\langle \text{bool exp} \rangle) \mid$
 $(\langle \text{bool exp} \rangle \text{ AND } \langle \text{bool exp} \rangle) \mid$
 $(\langle \text{bool exp} \rangle \text{ OR } \langle \text{bool exp} \rangle)$

First, Tokens!

The input expression
“(T AND NOT (T))”
consists of:

<u>Token Text</u>	<u>Token Kind</u>
"("	LEFT_PAREN
" "	WHITE_SPACE
"T"	TRUE_VALUE
" "	WHITE_SPACE
"AND"	AND_OPRTR
" "	WHITE_SPACE
"NOT"	NOT_OPRTR
" "	WHITE_SPACE
"("	LEFT_PAREN
" "	WHITE_SPACE
"T"	TRUE_VALUE
" "	WHITE_SPACE
")"	RIGHT_PAREN
" "	WHITE_SPACE
")"	RIGHT_PAREN

First Token?

- Ignore WHITE_SPACE tokens
- First non white-space token can only be one of:
 - T
 - F
 - NOT
 - (

What If First Token is T?

- What rewrite rule will be used to derive the Boolean expression?

- Draw the derivation tree:

First Token is T Continued...

- What is the *entire* expression?
- What is its value?

What If First Token is NOT?

- What will be the first rewrite rule used to derive the expression?
- Draw the top 2 levels of the derivation tree:
- How do we proceed?

What If First Token is (?)

- What will be the first rewrite rule used to derive the expression?
- Draw the top 2 levels of the derivation tree:
- How do we proceed?

Summing Up...

```
procedure Evaluate_Bool_Exp (  
  alters Text& input,  
  produces Boolean& result  
)  
{  
  object Token t, oprtr;  
  object Boolean left, right;  
  GetNextNonWSToken (input, t);  
  case_select (t.Kind ())  
  {  
    case FALSE_VALUE: {  
      result = false;  
    } break;  
    case TRUE_VALUE: {  
      result = true;  
    } break;  
  }  
}
```

```
case NOT_OPRTR: {  
  GetNextNonWSToken (input, t); // (  
  Evaluate_Bool_Exp (input, result);  
  GetNextNonWSToken (input, t); // )  
  result = not result;  
} break;  
case LEFT_PAREN: {  
  Evaluate_Bool_Exp (input, left);  
  GetNextNonWSToken (input, oprtr);  
  Evaluate_Bool_Exp (input, right);  
  GetNextNonWSToken (input, t); // )  
  if (oprtr.Kind () == AND_OPRTR)  
  {  
    result = left and right;  
  }  
  else // oprtr.Kind () == OR_OPRTR  
  {  
    result = left or right;  
  }  
} break;  
}
```

Processing Expressions: A Second Example

$\langle \text{expression} \rangle \rightarrow \langle \text{expression} \rangle \langle \text{addop} \rangle \langle \text{term} \rangle \mid \langle \text{term} \rangle$
 $\langle \text{term} \rangle \rightarrow \langle \text{term} \rangle \langle \text{multop} \rangle \langle \text{factor} \rangle \mid \langle \text{factor} \rangle$
 $\langle \text{factor} \rangle \rightarrow (\langle \text{expression} \rangle) \mid \langle \text{digit seqnce} \rangle$
 $\langle \text{addop} \rangle \rightarrow + \mid -$
 $\langle \text{multop} \rangle \rightarrow * \mid \text{DIV} \mid \text{MOD}$
 $\langle \text{digit seqnce} \rangle \rightarrow \langle \text{digit} \rangle \langle \text{digit seqnce} \rangle \mid \langle \text{digit} \rangle$
 $\langle \text{digit} \rangle \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

First, Left-Recursive Rules

- What's the problem?
- Replace the following rewrite rules:
 $\langle \text{expression} \rangle \rightarrow \langle \text{expression} \rangle \langle \text{addop} \rangle \langle \text{term} \rangle \mid \langle \text{term} \rangle$
 $\langle \text{term} \rangle \rightarrow \langle \text{term} \rangle \langle \text{multop} \rangle \langle \text{factor} \rangle \mid \langle \text{factor} \rangle$
- with the following rewrite rules:
 $\langle \text{expression} \rangle \rightarrow \langle \text{term} \rangle \{ \langle \text{addop} \rangle \langle \text{term} \rangle \}$
 $\langle \text{term} \rangle \rightarrow \langle \text{factor} \rangle \{ \langle \text{multop} \rangle \langle \text{factor} \rangle \}$

Revised CFG for Expressions

$\langle \text{expression} \rangle \rightarrow \langle \text{term} \rangle \{ \langle \text{addop} \rangle \langle \text{term} \rangle \}$
 $\langle \text{term} \rangle \rightarrow \langle \text{factor} \rangle \{ \langle \text{multop} \rangle \langle \text{factor} \rangle \}$
 $\langle \text{factor} \rangle \rightarrow (\langle \text{expression} \rangle) \mid \langle \text{digit seqnce} \rangle$
 $\langle \text{addop} \rangle \rightarrow + \mid -$
 $\langle \text{multop} \rangle \rightarrow * \mid \text{DIV} \mid \text{MOD}$
 $\langle \text{digit seqnce} \rangle \rightarrow \langle \text{digit} \rangle \langle \text{digit seqnce} \rangle \mid \langle \text{digit} \rangle$
 $\langle \text{digit} \rangle \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

Evaluating Expressions

- Recursive descent parser
- One operation per nonterminal symbol (for $\langle \text{expression} \rangle$, $\langle \text{term} \rangle$, $\langle \text{factor} \rangle$)
- Tokenizer breaks up input in tokens-- (Text, Integer) pairs
- Tokenizer also handles other nonterminal symbols ($\langle \text{addop} \rangle$, $\langle \text{multop} \rangle$, $\langle \text{digit seqnce} \rangle$, and $\langle \text{digit} \rangle$)

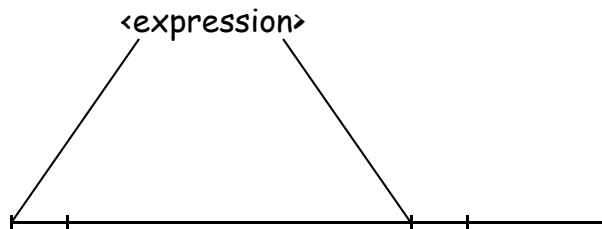
Evaluation Operation for Nonterminal <expression>

- How does the following operation work? (Check out the next slide)

```
global_procedure Evaluate_Expression (  
  alters Character_IStream& ins,  
  alters Text& token_text,  
  alters Integer& token_kind,  
  produces Integer& value  
);
```

Picture Specs for Evaluate_Expression

ins.content = "5 + 3 - 2 plus some more text"
token texts: "5", "+", "3", "-", "2", ...

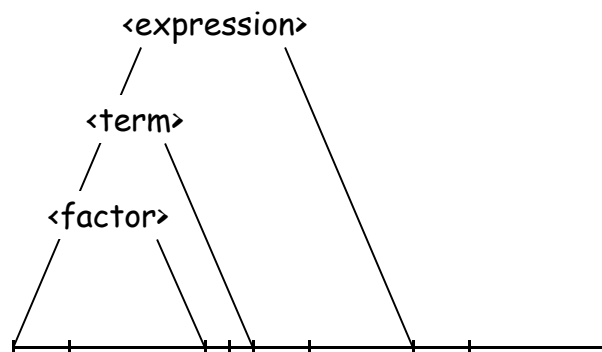


Evaluation Operation for Nonterminal <factor>

- How does the following operation work? (Check out the next slide)

```
global_procedure Evaluate_Factor (  
  alters Character_IStream& ins,  
  alters Text& token_text,  
  alters Integer& token_kind,  
  produces Integer& value  
);
```

Picture Specs for Evaluate_Factor



What About the Other Nonterminal Symbols?

- `<addop>`, `<multop>`, `<digit seqnce>`, and `<digit>` can be handled by the tokenizer
- However, in warm-up for closed lab:
 - no tokenizer
 - just deal with characters one at a time
 - use lookahead character
 - let the CFG drive the implementation of the operations.

How To Write Recursive Descent Parsers

1. One operation per nonterminal (except single-token nonterminals if using tokenizer)
2. nonterminal in rewrite rule → call operation to parse nonterminal
3. terminal (or single-token nonterminal) in rewrite rule → advance input (get next token)
4. `|` in rewrite rules → if-else-if in parser
5. `{}` in rewrite rules → loop in parser

Revised CFG for Expressions

$\langle \text{expression} \rangle \rightarrow \langle \text{term} \rangle \{ \langle \text{addop} \rangle \langle \text{term} \rangle \}$
 $\langle \text{term} \rangle \rightarrow \langle \text{factor} \rangle \{ \langle \text{multop} \rangle \langle \text{factor} \rangle \}$
 $\langle \text{factor} \rangle \rightarrow (\langle \text{expression} \rangle) \mid \langle \text{digit seqnce} \rangle$
 $\langle \text{addop} \rangle \rightarrow + \mid -$
 $\langle \text{multop} \rangle \rightarrow * \mid \text{DIV} \mid \text{MOD}$
 $\langle \text{digit seqnce} \rangle \rightarrow \langle \text{digit} \rangle \langle \text{digit seqnce} \rangle \mid \langle \text{digit} \rangle$
 $\langle \text{digit} \rangle \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

1. One operation per nonterminal (except single-token nonterminals)

- $\langle \text{expression} \rangle \rightarrow \text{Evaluate_Expression}$
- $\langle \text{term} \rangle \rightarrow \text{Evaluate_Term}$
- $\langle \text{factor} \rangle \rightarrow \text{Evaluate_Factor}$

- $\langle \text{addop} \rangle, \langle \text{multop} \rangle, \langle \text{digit seqnce} \rangle \rightarrow$
single-token nonterminals

*2. nonterminal in rewrite rule →
call operation to parse nonterminal*

`<expression> → <term> { <addop> <term> }`

```
procedure_body Evaluate_Expression (...)  
{  
    Evaluate_Term (...)  
    ...  
}
```

*2. nonterminal in rewrite rule →
call operation to parse nonterminal*

`<term> → <factor> { <multop> <factor> }`

```
procedure_body Evaluate_Term (...)  
{  
    Evaluate_Factor (...)  
    ...  
}
```

*3. terminal (or single-token nonterminal) in
rewrite rule → advance input (get token)*

`<factor> → (<expression>) | <digit seqnce>`

```
procedure_body Evaluate_Factor (...)  
{  
    ...  
    GetNextNonWSToken (...)  
    Evaluate_Expression (...)  
    ...  
}
```

*4. / in rewrite rules →
if-else-if in parser*

`<factor> → (<expression>) | <digit seqnce>`

```
procedure_body Evaluate_Factor (...)  
{  
    if (tk == LEFT_PAREN) {  
        GetNextNonWSToken (...)  
        Evaluate_Expression (...)  
        GetNextNonWSToken (...)  
    } else {  
        ...  
        GetNextNonWSToken (...)  
    }  
}
```

*5. {} in rewrite rules →
loop in parser*

<expression> → <term> { <addop> <term> }

```
procedure_body Evaluate_Expression (...)  
{  
    Evaluate_Term (...)  
    while ((tk == PLUS) or (tk == MINUS)) {  
        GetNextNonWSToken (...)  
        Evaluate_Term (...)  
        ...  
    }  
}
```

*5. {} in rewrite rules →
loop in parser*

<term> → <factor> { <multop> <factor> }

```
procedure_body Evaluate_Term (...)  
{  
    Evaluate_Factor (...)  
    while ((tk == STAR) or (tk == DIV) or (tk == MOD)) {  
        GetNextNonWSToken (...)  
        Evaluate_Factor (...)  
        ...  
    }  
}
```