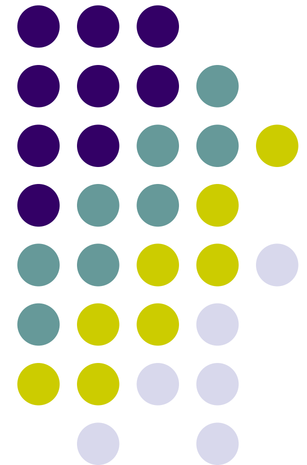
CS 460

Programming Languages

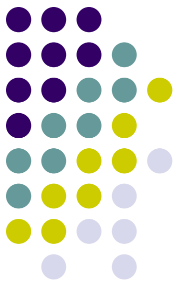
Fall 2023

Dr. Watts

(13 November 2023)

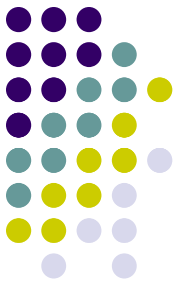


Assignments



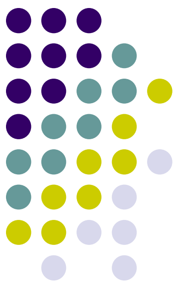
- Exercise 2
  - Script running so that your groups can improve their testing techniques
- Exercise 3
  - Your secret folder now contains the .dbg, .lst, .p1, and .p2 files for your lastnameE3.pl460 submission
- Exercise 4
  - .txt file due Wednesday, 2:30 pm
  - We will discuss this in class

Project 2



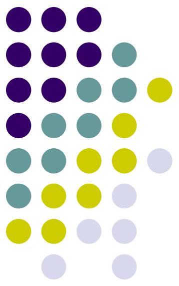
- Spec and Framework posted
- Extra Credit
  - PL460 program that uses all 93 grammar rules
  - Thursday, 16 November 2023, 6:59 am
  - Draw from Exercise 3
- Suggestions
  - Create First and Follow sets
  - Start with sets for the “Short Grammar”
  - Add in the remaining grammar rules
  - Testing

First and Follow Sets

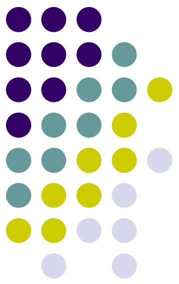


- Firsts
  - A terminal symbol T_i is a member of the First Set of non-terminal symbol <nt$_j$> if T_i can become the first terminal symbol in a complete expansion of <nt$_j$>.
- Follows
  - A terminal symbol T_i is a member of the Follow Set of non-terminal symbol <nt$_j$> if T_i can become the first terminal symbol immediately following a complete expansion of <nt$_j$>.

Short Project Example

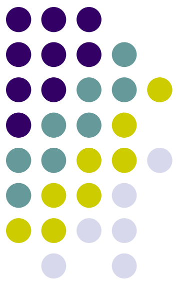


7. <stmt> -> <literal>
8. <stmt> -> IDENT_T
9. <stmt> -> LPAREN_T <action> RPAREN_T
10. <literal> -> NUMLIT_T
11. <literal> -> STRLIT_T
12. <literal> -> <logical_lit>
13. <logical_lit> -> TRUE_T
14. <logical_lit> -> FALSE_T



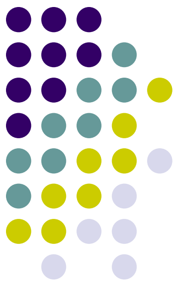
```
void SyntacticalAnalyzer::stmt ()
{
	if (token == NUMLIT_T || token == TRUE_T || token == FALSE_T
	                      || token == STRLIT_T)
	{ // Rule 7  <stmt> -> <literal>
		lex->ruleFile << "Using Rule 7\n";
		literal ();
	}
	else if (token == IDENT_T)
	{ // Rule 8  <stmt> -> IDENT_T
		lex->ruleFile << "Using Rule 8\n";
		token = lex->GetToken ();
	}
	else if (token == LPAREN_T)
	{ // Rule 9  <stmt> -> LPAREN_T <action> RPAREN_T
		lex->ruleFile << "Using Rule 9\n";
		token = lex->GetToken ();
		action ();
		if (token == RPAREN_T)
			token = lex->GetToken ();
		else
			lex->ReportError ("( expected");
	}
	else
		lex->ReportError (lex->GetLexeme() + " unexpected");
	return;
}
```

Parser Functions



- “Expectations”
- Preconditions?
- Postconditions?
- Preconditions for stmt ()?
- Postconditions for stmt ()?

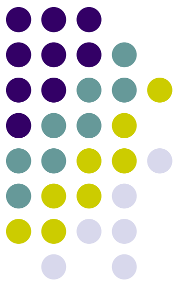
Firsts and Follows for <stmt>



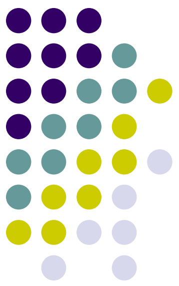
```
set <int> firsts {NUMLIT_T, TRUE_T, FALSE_T,
      STRLIT_T, IDENT_T, LPAREN_T, EOF_T};

set <int> follows {NUMLIT_T, TRUE_T, FALSE_T,
      LPAREN_T, RPAREN_T, IDENT_T, STRLIT_T, EOF_T};
```

Error Recovery Methods

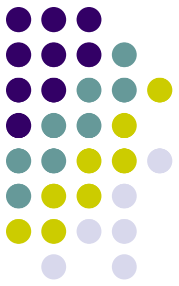


- At the beginning of each function
  - Ensure that the current token is an element of the firsts of the non-terminal
- At the end of each function
  - Ensure that the current token is an element of the follows of the non-terminal
- GetToken loop
  - GetToken until current token is an element of the set
  - Issue error messages



```
void SyntacticalAnalyzer::stmt ()
{
	// At beginning . . .
	if (token == NUMLIT_T || token == TRUE_T || token == FALSE_T || token == STRLIT_T)
	{ // Rule 7  <stmt> -> <literal>
		lex->ruleFile << "Using Rule 7\n";
		literal ();
	}
	else if (token == IDENT_T)
	{ // Rule 8  <stmt> -> IDENT_T
		lex->ruleFile << "Using Rule 8\n";
		token = lex->GetToken ();
	}
	else if (token == LPAREN_T)
	{ // Rule 9  <stmt> -> LPAREN_T <action> RPAREN_T
		lex->ruleFile << "Using Rule 9\n";
		token = lex->GetToken ();
		action ();
		if (token == RPAREN_T)
			token = lex->GetToken ();
		else
			lex->ReportError ("( expected");
	}
	else
		lex->ReportError (lex->GetLexeme() + " unexpected");
	// At end - before return . . .
	return;
}
```

Project 2 Testing (P2Tests)



No errors

```
(define (square n)
        (* n n)
)


(define (main)
        (display (square (read)))
        (newline)
)


(main)
```

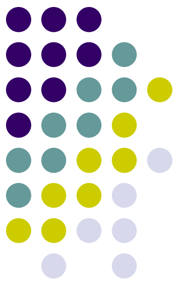
Add an error

```
(define (square n)
        (* n n)
)


(define (main)
        (display (square (read)))
        (newline)
)


(main)
```

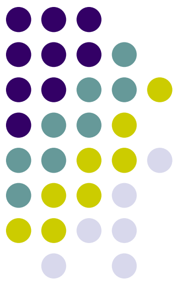
Recursion in PL460



- Rules of Recursion - Determine
  - the unit of work
  - the base case (how to make it stop)
  - the external call (how to make it start)
  - the internal call(s) (how to keep it going)

- PL460 Examples

Tail Recursion Example



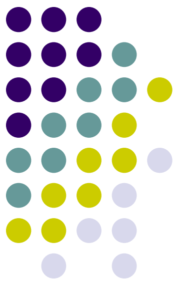
```
(define (tailRec value)
        (display value) (display " ")
        (if (> value 1)
                (tailRec (/ value 2))
        )
)

(define (main)
        (display "(tailRec 8) --> ")
        (tailRec 8)
        (newline)
        (display "(tailRec 30) --> ")
        (tailRec 30)
        (newline)
)

(main)
```

- (tailRec 8) --> 8 4 2 1
- (tailRec 30) --> 30 15 15/2 15/4 15/8 15/16

Head Recursion Example



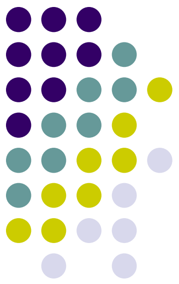
```
(define (headRec value)
        (if (> value 1)
                (headRec (/ value 2))
        )
        (display value) (display " ")
)

(define (main)
        (display "(headRec 8) --> ")
        (headRec 8)
        (newline)
        (display "(headRec 30) --> ")
        (headRec 30)
        (newline)
)

(main)
```

- `(headRec 8) --> 1 2 4 8`
- `(headRec 30) --> 15/16 15/8 15/4 15/2 15 30`

“Middle” Recursion Example



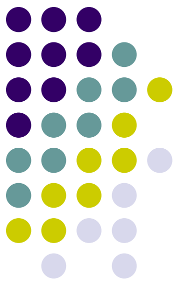
```
(define (midRec value)
        (display value) (display " ")
        (if (> value 1)
                (midRec (/ value 2))
        )
        (display value) (display " ")
)

(define (main)
        (display "(midRec 8) --> ")
        (midRec 8)
        (newline)
        (display "(midRec 30) --> ")
        (midRec 30)
        (newline)
)

(main)
```

- `(midRec 8) --> 8 4 2 1 1 2 4 8`
- `(midRec 30) --> 30 15 15/2 15/4 15/8 15/16 15/16 15/8 15/4 15/2 15 30`

Head and Tail Recursion

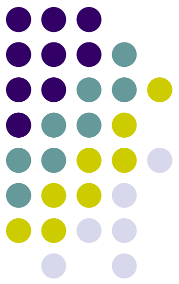


```
(define (headTailRec value)
        (if (> value 1)
                (headTailRec (/ value 2))
        )
        (display value) (display " ")
        (if (> value 1)
                (headTailRec (/ value 2))
        )
)

(define (main)
        (display "(headTailRec 8) --> ")
        (headTailRec 8)
        (newline)
        (display "(headTailRec 16) --> ")
        (headTailRec 16)
        (newline)
)

(main)
```

Head and Tail Recursion



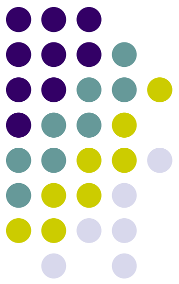
```
(define (headTailRec value)
          (if (> value 1)
                    (headTailRec (/ value 2))
          )
          (display value) (display " ")
          (if (> value 1)
                    (headTailRec (/ value 2))
          )
)

(define (main)
          (display "(headTailRec 8) --> ")
          (headTailRec 8)
          (newline)
          (display "(headTailRec 16) --> ")
          (headTailRec 16)
          (newline)
)

(main)
```

- `(headTailRec 8) --> 1 2 1 4 1 2 1 8 1 2 1 4 1 2 1`
- `(headTailRec 16) --> 1 2 1 4 1 2 1 8 1 2 1 4 1 2 1 16 1 2 1 4 1 2 1 8 1 2 1 4 1 2 1`

Head, Middle andTail Recursion

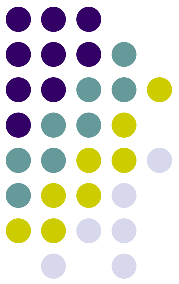


```
(define (headMidTailRec value)
        (if (> value 1)
                (headMidTailRec (/ value 2))
        )
        (display value) (display " ")
        (if (> value 1)
                (headMidTailRec (/ value 2))
        )
        (display value) (display " ")
        (if (> value 1)
                (headMidTailRec (/ value 2))
        )
)

(define (main)
        (display "(headMidTailRec 8) --> ")
        (headMidTailRec 8)
        (newline)
)

(main)
```

Head, Middle andTail Recursion



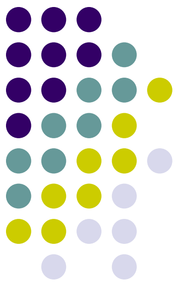
```
(define (headMidTailRec value)
          (if (> value 1)
                    (headMidTailRec (/ value 2))
          )
          (display value) (display " ")
          (if (> value 1)
                    (headMidTailRec (/ value 2))
          )
          (display value) (display " ")
          (if (> value 1)
                    (headMidTailRec (/ value 2))
          )
)

(define (main)
          (display "(headMidTailRec 8) --> ")
          (headMidTailRec 8)
          (newline)
)

(main)
```

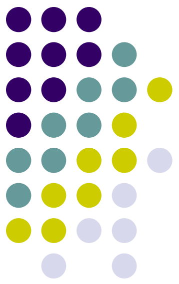
- (headMidTailRec 8) --> 1 1 2 1 1 2 1 1 4 1 1 2 1 1 2 1 1 4 1 1 2 1 1 2 1 1 8 1 1 2 1 1 2 1 1 4 1 1 2 1 1 2 1 1 4 1 1 2 1 1 2 1 1 8 1 1 2 1 1 2 1 1 4 1 1 2 1 1 2 1 1 4 1 1 2 1 1 2 1 1

Expressions and Assignment Statements (Chapter 7)



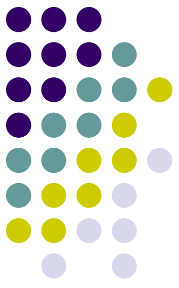
- Arithmetic Expressions
- Overloaded Operators
- Type Conversions
- Relational and Boolean Expressions
- Short-Circuit Evaluation
- Assignment Statements
- Mixed-Mode Assignment

Arithmetic Expressions



- Operators
- Operator Evaluation Order
  - Precedence
  - Commutativity
  - Associativity
  - Parenthesis
  - Conditional Expressions
  - Operand Evaluation Order
    - Side Effects

Overloaded Operators – Ex 5



- `money operator + (const money & M) const;`
- `money operator += (const money & M);`
- `money operator - (const money & M) const;`
- `money operator -= (const money & M);`
- `money operator * (const double & F) const;`
- **`friend money operator * (const double & Factor, const money & M);`**
- `money operator *= (const double & Factor);`
- `money operator / (const double & Divisor) const;`
- `money operator /= (const double & Divisor);`
- `money operator % (const int & Divisor) const;`
- `money operator %= (const int & Divisor);`
- `money operator ++ (); // Pre increment`
- `money operator ++ (int); // Post increment`
- `money operator -- (); // Pre decrement`
- `money operator -- (int); // Post decrement`

- `bool operator == (const money & M) const;`
- `bool operator != (const money & M) const;`
- `bool operator < (const money & M) const;`
- `bool operator <= (const money & M) const;`
- `bool operator > (const money & M) const;`
- `bool operator >= (const money & M) const;`