Parsing Strategies

Eric Roberts
CS 106B
March 4, 2015

The Problem of Parsing

- The rules for forming an expression can be expressed in the form of a ***grammar***, as follows:

$$\mathbf{E} \rightarrow \textit{constant}$$
$$\mathbf{E} \rightarrow \textit{identifier}$$
$$\mathbf{E} \rightarrow \mathbf{E}\ \textit{op}\ \mathbf{E}$$
$$\mathbf{E} \rightarrow (\ \mathbf{E}\ )$$

- The process of translating an expression from a string to its internal form is called ***parsing***.

A Two-Level Grammar

- The problem of parsing an expression can be simplified by changing the grammar to one that has two levels:
  - An ***expression*** is either a *term* or two expressions joined by an operator.
  - A ***term*** is either a constant, an identifier, or an expression enclosed in parentheses.
- This design is reflected in the following revised grammar.

E → **T**

E → **E** *op* **E**

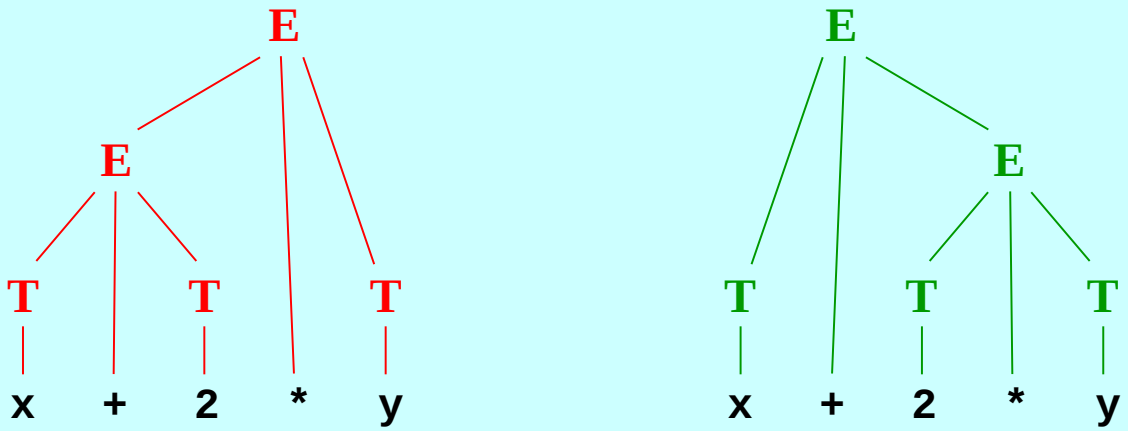
T → *constant*

T → *identifier*

T → **(E)**

Ambiguity in Parse Structures

- Although the two-level grammar from the preceding slide can recognize any expression, it is ***ambiguous*** because the same input string can generate more than one parse tree.

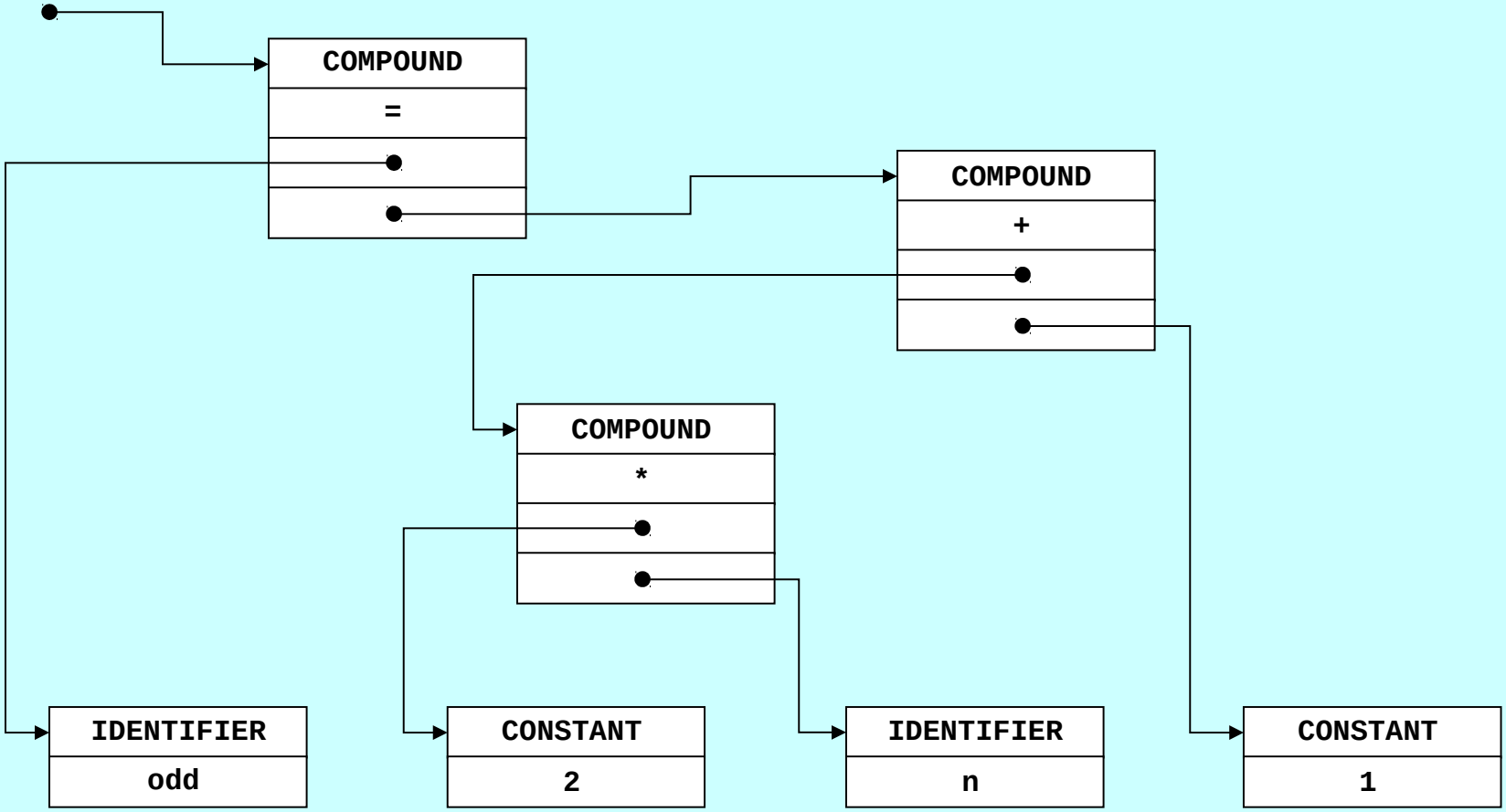


- Ambiguity in grammars is typically resolved by providing the parser with information about the ***precedence*** of the operators.

Exercise: Parsing an Expression

- Diagram the expression tree that results from the input string

odd = 2 * n + 1



The **parser.cpp** Implementation

```
/*
 * Implementation notes: readE
 * Usage: exp = readE(scanner, prec);
 * ----------------------------------
 * This function reads the next expression from the scanner by
 * matching the input to the following ambiguous grammar:
 *
 *       E  ->  T
 *       E  ->  E op E
 *
 * This version of the method uses precedence to resolve ambiguity.
 */

Expression *readE(TokenScanner & scanner, int prec) {
   Expression *exp = readT(scanner);
   string token;
   while (true) {
      token = scanner.nextToken();
      int tprec = precedence(token);
      if (tprec <= prec) break;
      Expression *rhs = readE(scanner, tprec);
      exp = new CompoundExp(token, exp, rhs);
   }
   scanner.saveToken(token);
   return exp;
}
```

The `parser.cpp` Implementation

```
/*
 * Function: readT
 * Usage: exp = readT(scanner);
 * ----------------------------
 * This function reads a single term from the scanner.
 */

Expression *readT(TokenScanner & scanner) {
   string token = scanner.nextToken();
   TokenType type = scanner.getTokenType(token);
   if (type == WORD) return new IdentifierExp(token);
   if (type == NUMBER) return new ConstantExp(stringToInteger(token));
   if (token != "(") error("Illegal term in expression");
   Expression *exp = readE(scanner, 0);
   if (scanner.nextToken() != ")") {
      error("Unbalanced parentheses in expression");
   }
   return exp;
}
```

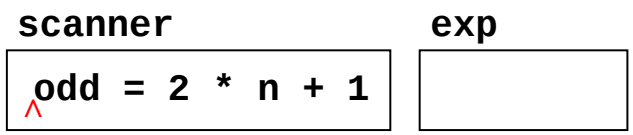
The **parser.cpp** Implementation

```
/*
 * Function: precedence
 * Usage: prec = precedence(token);
 * --------------------------------
 * This function returns the precedence of the specified operator
 * token.  If the token is not an operator, precedence returns 0.
 */

int precedence(string token) {
   if (token == "=") return 1;
   if (token == "+" || token == "-") return 2;
   if (token == "*" || token == "/") return 3;
   return 0;
}
```

Tracing the Precedence Parser

```
int main() {
   TokenScanner scanner = new TokenScanner();
   scanner.setInput("odd = 2 * n + 1");
   scanner.ignoreWhitespace();
   scanner.scanNumbers();
   Expression *exp = readE(scanner, 0);
   . . .
}
```



skip simulation

Tracing the Precedence Parser

```
int main() {
```

```
Expression *readE(TokenScanner & scanner, int prec) {
   Expression *exp = readT(scanner);
   string token;
   while (true) {
      token = scanner.nextToken();
      int tprec = precedence(token);
      if (tprec <= prec) break;
      Expression *rhs = readE(scanner, tprec);
      exp = new CompoundExp(token, exp, rhs);
   }
   scanner.saveToken(token);
   return exp;
}
```

scanner	prec	tprec	token	exp	rhs
odd = 2 * n + 1 ^	0				

Tracing the Precedence Parser

```
int main() {
```

```
Expression *readE(TokenScanner & scanner, int prec) {
```

```
Expression *readT(TokenScanner & scanner) {
   string token = scanner.nextToken();
   TokenType type = scanner.getTokenType(token);
   if (type == WORD) return new IdentifierExp(token);
   if (type == NUMBER) return new ConstantExp(stringToInteger(token));
   if (token != "(") error("Illegal term in expression");
   Expression *exp = readE(scanner, 0);
   if (scanner.nextToken() != ")") {
      error("Unbalanced parentheses in expression");
   }
   return exp;
}
```

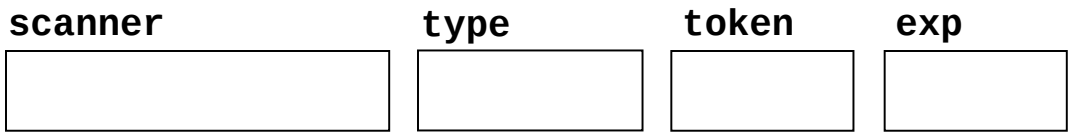
scanner	type	token	exp
odd = 2 * n + 1	WORD	odd	

Tracing the Precedence Parser

```
int main() {
```

```
Expression *readE(TokenScanner & scanner, int prec) {
```

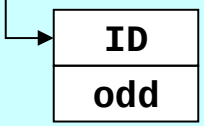
```
Expression *readT(TokenScanner & scanner) {
   string token = scanner.nextToken();
   TokenType type = scanner.getTokenType(token);
   if (type == WORD) return new IdentifierExp(token);
   if (type == NUMBER) return new ConstantExp(stringToInteger(token));
   if (token != "(") error("Illegal term in expression");
   Expression *exp = readE(scanner, 0);
   if (scanner.nextToken() != ")") {
      error("Unbalanced parentheses in expression");
   }
   return exp;
}
```



Tracing the Precedence Parser

```
int main() {
  Expression *readE(TokenScanner & scanner, int prec) {
     Expression *exp = readT(scanner);
     string token;
     while (true) {
        token = scanner.nextToken();
        int tprec = precedence(token);
        if (tprec <= prec) break;
        Expression *rhs = readE(scanner, tprec);
        exp = new CompoundExp(token, exp, rhs);
     }
     scanner.saveToken(token);
     return exp;
  }
```

scanner	prec	tprec	token	exp	rhs
odd = 2 * n + 1	0	1	=	•	



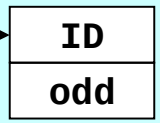
Tracing the Precedence Parser

```
int main() {
```

```
Expression *readE(TokenScanner & scanner, int prec) {
```

```
Expression *readE(TokenScanner & scanner, int prec) {
   Expression *exp = readT(scanner);
   string token;
   while (true) {
      token = scanner.nextToken();
      int tprec = precedence(token);
      if (tprec <= prec) break;
      Expression *rhs = readE(scanner, tprec);
      exp = new CompoundExp(token, exp, rhs);
   }
   scanner.saveToken(token);
   return exp;
}
```

scanner	prec	tprec	token	exp	rhs
odd = 2 * n + 1	1				



Tracing the Precedence Parser

```
int main() {
 Expression *readE(TokenScanner & scanner, int prec) {
  Expression *readE(TokenScanner & scanner, int prec) {
   Expression *readT(TokenScanner & scanner) {
      string token = scanner.nextToken();
      TokenType type = scanner.getTokenType(token);
      if (type == WORD) return new IdentifierExp(token);
      if (type == NUMBER) return new ConstantExp(stringToInteger(token));
      if (token != "(") error("Illegal term in expression");
      Expression *exp = readE(scanner, 0);
      if (scanner.nextToken() != ")") {
         error("Unbalanced parentheses in expression");
      }
      return exp;
   }
```

scanner	type	token	exp
odd = 2 * n + 1	NUMBER	2	

ID
odd

Tracing the Precedence Parser

```
int main() {
 Expression *readE(TokenScanner & scanner, int prec) {
  Expression *readE(TokenScanner & scanner, int prec) {
   Expression *readT(TokenScanner & scanner) {
      string token = scanner.nextToken();
      TokenType type = scanner.getTokenType(token);
      if (type == WORD) return new IdentifierExp(token);
      if (type == NUMBER) return new ConstantExp(stringToInteger(token));
      if (token != "(") error("Illegal term in expression");
      Expression *exp = readE(scanner, 0);
      if (scanner.nextToken() != ")") {
         error("Unbalanced parentheses in expression");
      }
      return exp;
   }
```

scanner	type	token	exp

ID
odd

Tracing the Precedence Parser

```
int main() {
```

```
Expression *readE(TokenScanner & scanner, int prec) {
```

```
Expression *readE(TokenScanner & scanner, int prec) {
   Expression *exp = readT(scanner);
   string token;
   while (true) {
      token = scanner.nextToken();
      int tprec = precedence(token);
      if (tprec <= prec) break;
      Expression *rhs = readE(scanner, tprec);
      exp = new CompoundExp(token, exp, rhs);
   }
   scanner.saveToken(token);
   return exp;
}
```

scanner	prec	tprec	token	exp	rhs
odd = 2 ^ * ^ n + 1	1	3	*	●	

Tracing the Precedence Parser

```
int main() {
```

```
Expression *readE(TokenScanner & scanner, int prec) {
```

```
Expression *readE(TokenScanner & scanner, int prec) {
```

```
Expression *readE(TokenScanner & scanner, int prec) {
   Expression *exp = readT(scanner);
   string token;
   while (true) {
      token = scanner.nextToken();
      int tprec = precedence(token);
      if (tprec <= prec) break;
      Expression *rhs = readE(scanner, tprec);
      exp = new CompoundExp(token, exp, rhs);
   }
   scanner.saveToken(token);
   return exp;
}
```

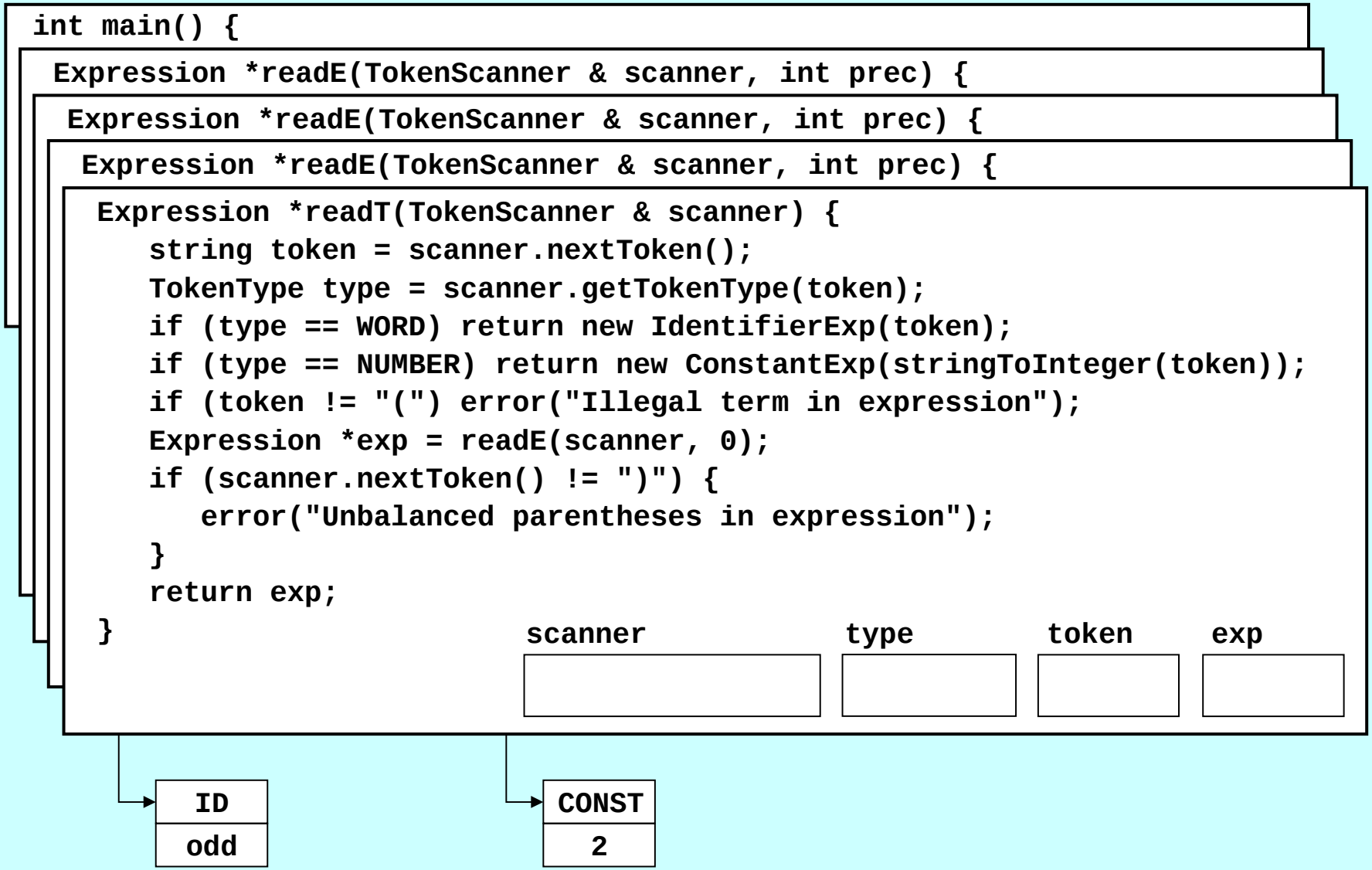
scanner	prec	tprec	token	exp	rhs
odd = 2 * n + 1	3				

Tracing the Precedence Parser

```
int main() {
 Expression *readE(TokenScanner & scanner, int prec) {
  Expression *readE(TokenScanner & scanner, int prec) {
   Expression *readE(TokenScanner & scanner, int prec) {
    Expression *readT(TokenScanner & scanner) {
       string token = scanner.nextToken();
       TokenType type = scanner.getTokenType(token);
       if (type == WORD) return new IdentifierExp(token);
       if (type == NUMBER) return new ConstantExp(stringToInteger(token));
       if (token != "(") error("Illegal term in expression");
       Expression *exp = readE(scanner, 0);
       if (scanner.nextToken() != ")") {
          error("Unbalanced parentheses in expression");
       }
       return exp;
    }
```

scanner	type	token	exp
odd = 2 * n + 1	WORD	n	

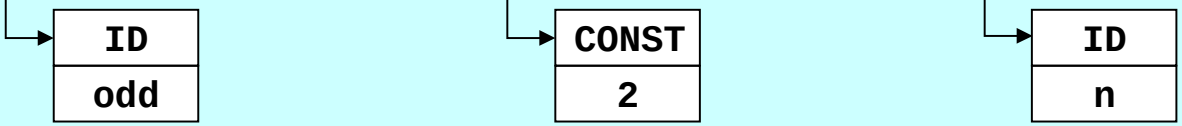
Tracing the Precedence Parser



Tracing the Precedence Parser

```
int main() {
 Expression *readE(TokenScanner & scanner, int prec) {
  Expression *readE(TokenScanner & scanner, int prec) {
   Expression *readE(TokenScanner & scanner, int prec) {
      Expression *exp = readT(scanner);
      string token;
      while (true) {
         token = scanner.nextToken();
         int tprec = precedence(token);
         if (tprec <= prec) break;
         Expression *rhs = readE(scanner, tprec);
         exp = new CompoundExp(token, exp, rhs);
      }
      scanner.saveToken(token);
      return exp;
   }
```

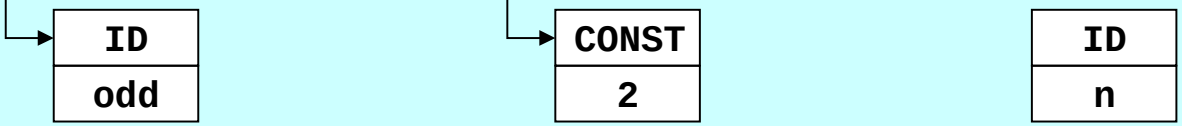
scanner	prec	tprec	token	exp	rhs
odd = 2 * n^+^1	3	2	+	•	



Tracing the Precedence Parser

```
int main() {
 Expression *readE(TokenScanner & scanner, int prec) {
  Expression *readE(TokenScanner & scanner, int prec) {
   Expression *readE(TokenScanner & scanner, int prec) {
      Expression *exp = readT(scanner);
      string token;
      while (true) {
         token = scanner.nextToken();
         int tprec = precedence(token);
         if (tprec <= prec) break;
         Expression *rhs = readE(scanner, tprec);
         exp = new CompoundExp(token, exp, rhs);
      }
      scanner.saveToken(token);
      return exp;
   }
```

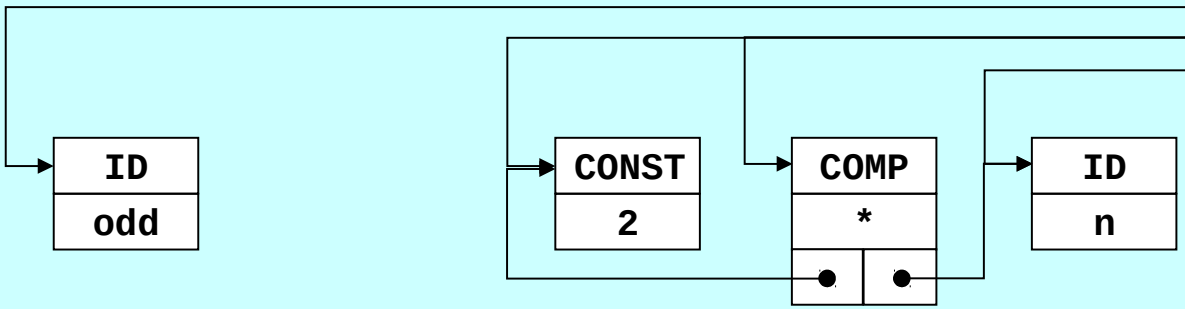
scanner	prec	tprec	token	exp	rhs



Tracing the Precedence Parser

```
int main() {
 Expression *readE(TokenScanner & scanner, int prec) {
  Expression *readE(TokenScanner & scanner, int prec) {
     Expression *exp = readT(scanner);
     string token;
     while (true) {
        token = scanner.nextToken();
        int tprec = precedence(token);
        if (tprec <= prec) break;
        Expression *rhs = readE(scanner, tprec);
        exp = new CompoundExp(token, exp, rhs);
     }
     scanner.saveToken(token);
     return exp;
  }
```

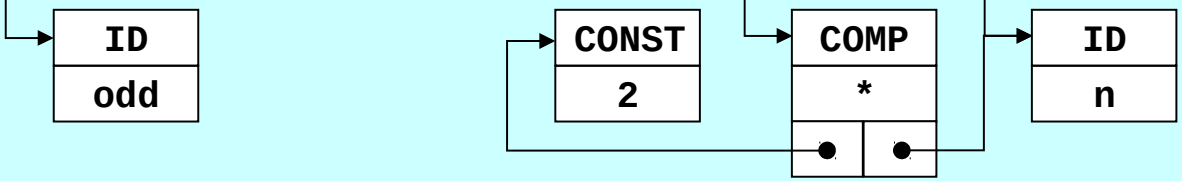
scanner	prec	tprec	token	exp	rhs
odd = 2 * n ^ + ^ 1	1	2	+		



Tracing the Precedence Parser

```
int main() {
Expression *readE(TokenScanner & scanner, int prec) {
Expression *readE(TokenScanner & scanner, int prec) {
Expression *readE(TokenScanner & scanner, int prec) {
   Expression *exp = readT(scanner);
   string token;
   while (true) {
      token = scanner.nextToken();
      int tprec = precedence(token);
      if (tprec <= prec) break;
      Expression *rhs = readE(scanner, tprec);
      exp = new CompoundExp(token, exp, rhs);
   }
   scanner.saveToken(token);
   return exp;
}
```

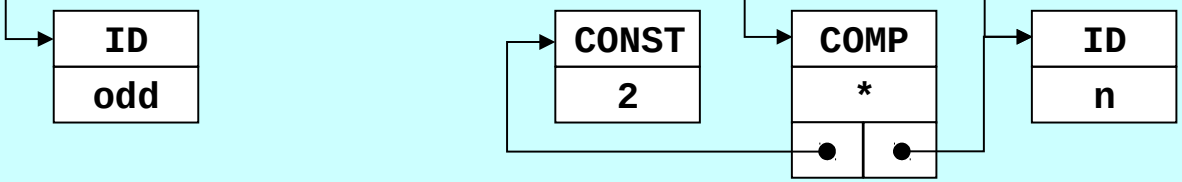
scanner	prec	tprec	token	exp	rhs
odd = 2 * n + 1	2				



Tracing the Precedence Parser

```
int main() {
 Expression *readE(TokenScanner & scanner, int prec) {
  Expression *readE(TokenScanner & scanner, int prec) {
   Expression *readE(TokenScanner & scanner, int prec) {
    Expression *readT(TokenScanner & scanner) {
       string token = scanner.nextToken();
       TokenType type = scanner.getTokenType(token);
       if (type == WORD) return new IdentifierExp(token);
       if (type == NUMBER) return new ConstantExp(stringToInteger(token));
       if (token != "(") error("Illegal term in expression");
       Expression *exp = readE(scanner, 0);
       if (scanner.nextToken() != ")") {
          error("Unbalanced parentheses in expression");
       }
       return exp;
    }
```

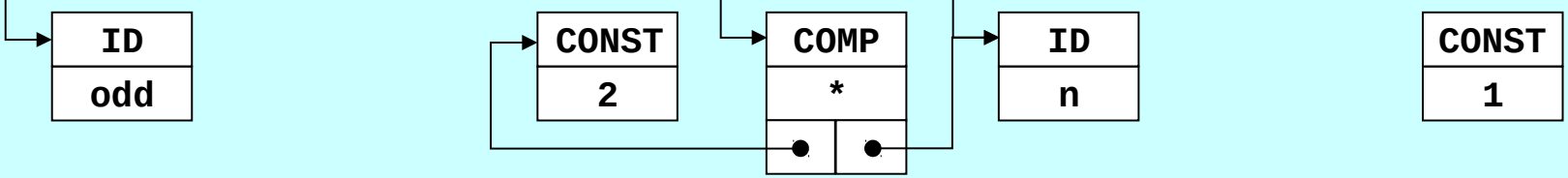
scanner	type	token	exp
odd = 2 * n + 1	NUMBER	1	



Tracing the Precedence Parser

```
int main() {
 Expression *readE(TokenScanner & scanner, int prec) {
  Expression *readE(TokenScanner & scanner, int prec) {
   Expression *readE(TokenScanner & scanner, int prec) {
    Expression *readT(TokenScanner & scanner) {
       string token = scanner.nextToken();
       TokenType type = scanner.getTokenType(token);
       if (type == WORD) return new IdentifierExp(token);
       if (type == NUMBER) return new ConstantExp(stringToInteger(token));
       if (token != "(") error("Illegal term in expression");
       Expression *exp = readE(scanner, 0);
       if (scanner.nextToken() != ")") {
          error("Unbalanced parentheses in expression");
       }
       return exp;
    }
```

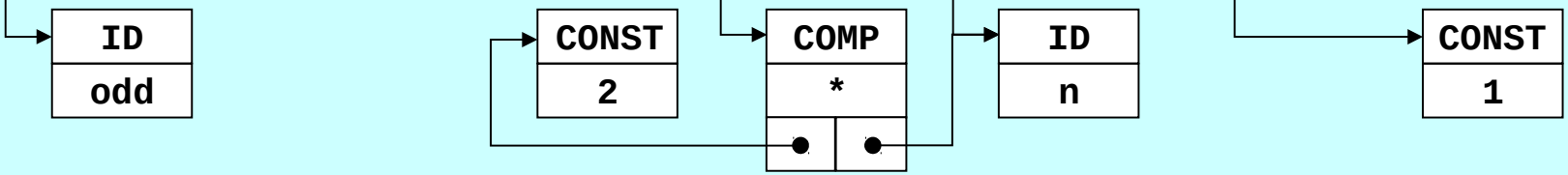
scanner | type | token | exp



Tracing the Precedence Parser

```
int main() {
Expression *readE(TokenScanner & scanner, int prec) {
Expression *readE(TokenScanner & scanner, int prec) {
Expression *readE(TokenScanner & scanner, int prec) {
   Expression *exp = readT(scanner);
   string token;
   while (true) {
      token = scanner.nextToken();
      int tprec = precedence(token);
      if (tprec <= prec) break;
      Expression *rhs = readE(scanner, tprec);
      exp = new CompoundExp(token, exp, rhs);
   }
   scanner.saveToken(token);
   return exp;
}
```

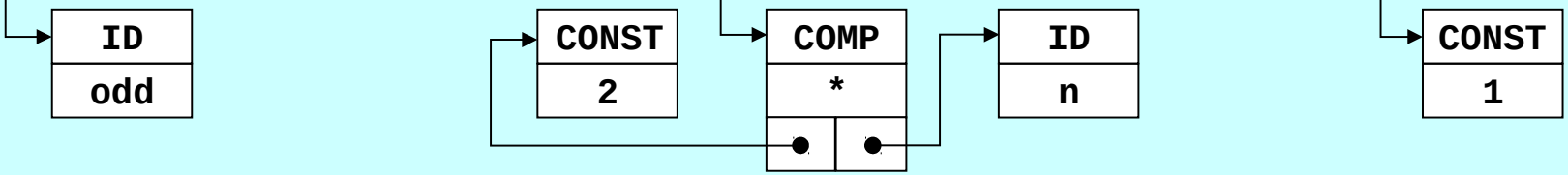
scanner	prec	tprec	token	exp	rhs
odd = 2 * n + 1 ^	2	0		•	



Tracing the Precedence Parser

```
int main() {
Expression *readE(TokenScanner & scanner, int prec) {
Expression *readE(TokenScanner & scanner, int prec) {
Expression *readE(TokenScanner & scanner, int prec) {
   Expression *exp = readT(scanner);
   string token;
   while (true) {
      token = scanner.nextToken();
      int tprec = precedence(token);
      if (tprec <= prec) break;
      Expression *rhs = readE(scanner, tprec);
      exp = new CompoundExp(token, exp, rhs);
   }
   scanner.saveToken(token);
   return exp;
}
```

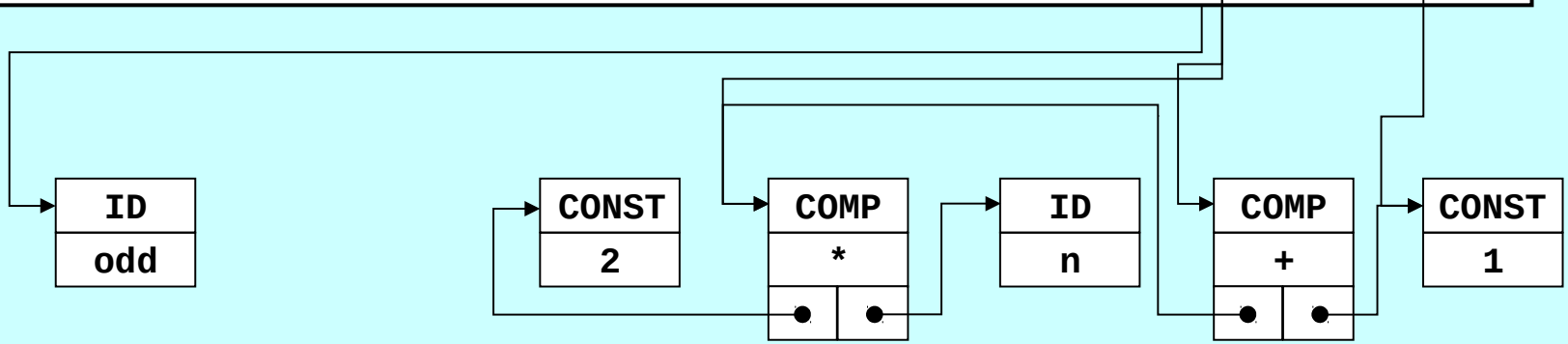
scanner | prec | tprec | token | exp | rhs



Tracing the Precedence Parser

```
int main() {
 Expression *readE(TokenScanner & scanner, int prec) {
  Expression *readE(TokenScanner & scanner, int prec) {
     Expression *exp = readT(scanner);
     string token;
     while (true) {
        token = scanner.nextToken();
        int tprec = precedence(token);
        if (tprec <= prec) break;
        Expression *rhs = readE(scanner, tprec);
        exp = new CompoundExp(token, exp, rhs);
     }
     scanner.saveToken(token);
     return exp;
  }
```

scanner	prec	tprec	token	exp	rhs
odd = 2 * n + 1 ^	1	0			



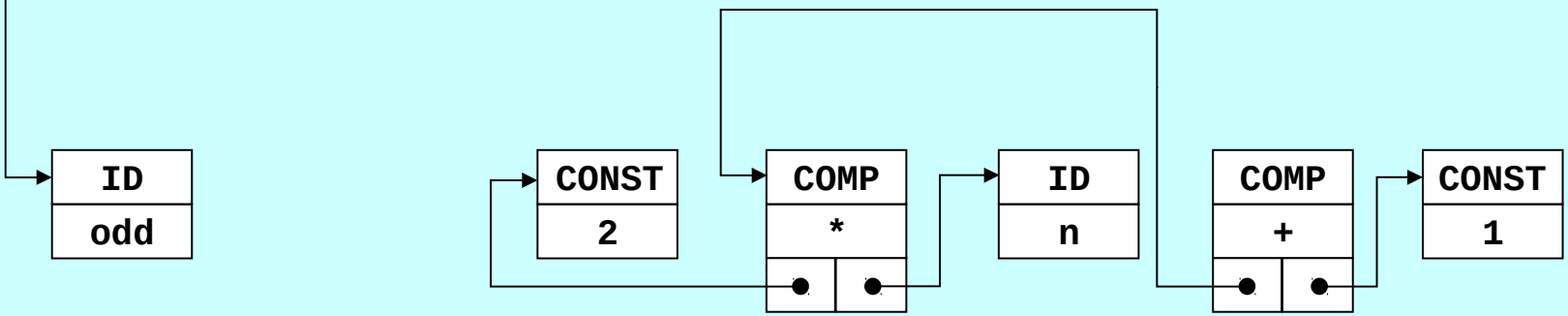
Tracing the Precedence Parser

```
int main() {
```

```
Expression *readE(TokenScanner & scanner, int prec) {
```

```
Expression *readE(TokenScanner & scanner, int prec) {
   Expression *exp = readT(scanner);
   string token;
   while (true) {
      token = scanner.nextToken();
      int tprec = precedence(token);
      if (tprec <= prec) break;
      Expression *rhs = readE(scanner, tprec);
      exp = new CompoundExp(token, exp, rhs);
   }
   scanner.saveToken(token);
   return exp;
}
```

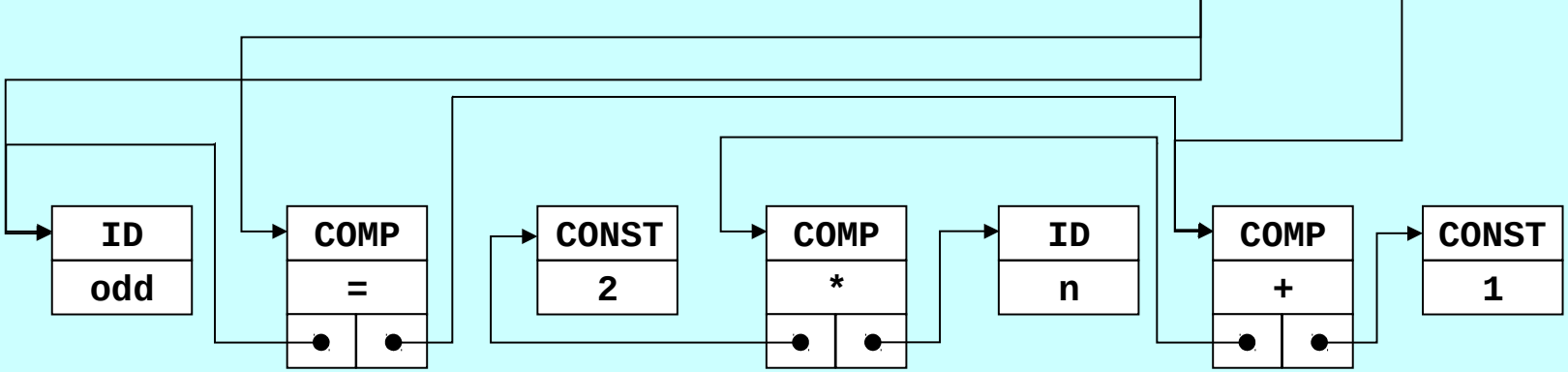
scanner	prec	tprec	token	exp	rhs



Tracing the Precedence Parser

```
int main() {
  Expression *readE(TokenScanner & scanner, int prec) {
     Expression *exp = readT(scanner);
     string token;
     while (true) {
        token = scanner.nextToken();
        int tprec = precedence(token);
        if (tprec <= prec) break;
        Expression *rhs = readE(scanner, tprec);
        exp = new CompoundExp(token, exp, rhs);
     }
     scanner.saveToken(token);
     return exp;
  }
```

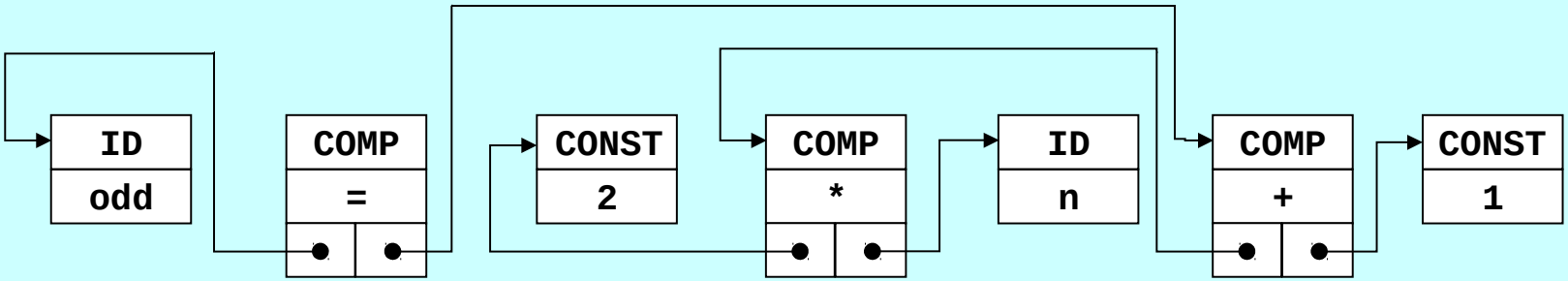
scanner	prec	tprec	token	exp	rhs
odd = 2 * n + 1 ^	0	0		•	•



Tracing the Precedence Parser

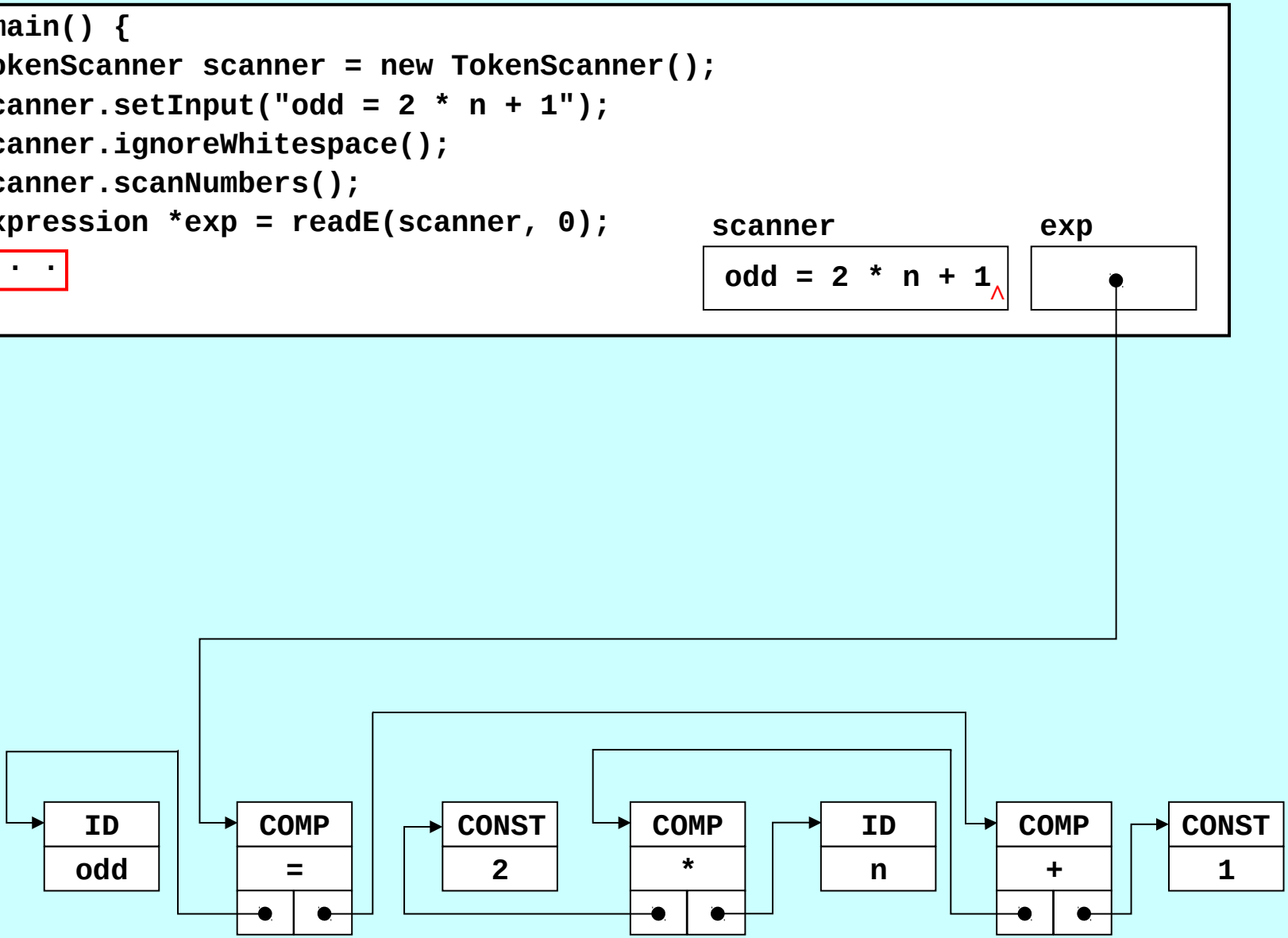
```
int main() {
 Expression *readE(TokenScanner & scanner, int prec) {
    Expression *exp = readT(scanner);
    string token;
    while (true) {
       token = scanner.nextToken();
       int tprec = precedence(token);
       if (tprec <= prec) break;
       Expression *rhs = readE(scanner, tprec);
       exp = new CompoundExp(token, exp, rhs);
    }
    scanner.saveToken(token);
    return exp;
 }
```

scanner	prec	tprec	token	exp	rhs



Tracing the Precedence Parser

```
int main() {
   TokenScanner scanner = new TokenScanner();
   scanner.setInput("odd = 2 * n + 1");
   scanner.ignoreWhitespace();
   scanner.scanNumbers();
   Expression *exp = readE(scanner, 0);
   . . .
}
```



To: BASIC Development Group

From: Bill Gates

Subject: C++ reimplementation

Date: April 1, 1981

This guy from Bell Labs, Bjarne Stroustroup, just sent me a parser program written in a new language he's calling C++. His parser is much simpler than ours but still seems quite efficient. The code is much easier to read as well.

I think it's time to move away from assembly language for our version of BASIC, and C++ may be just the right tool.

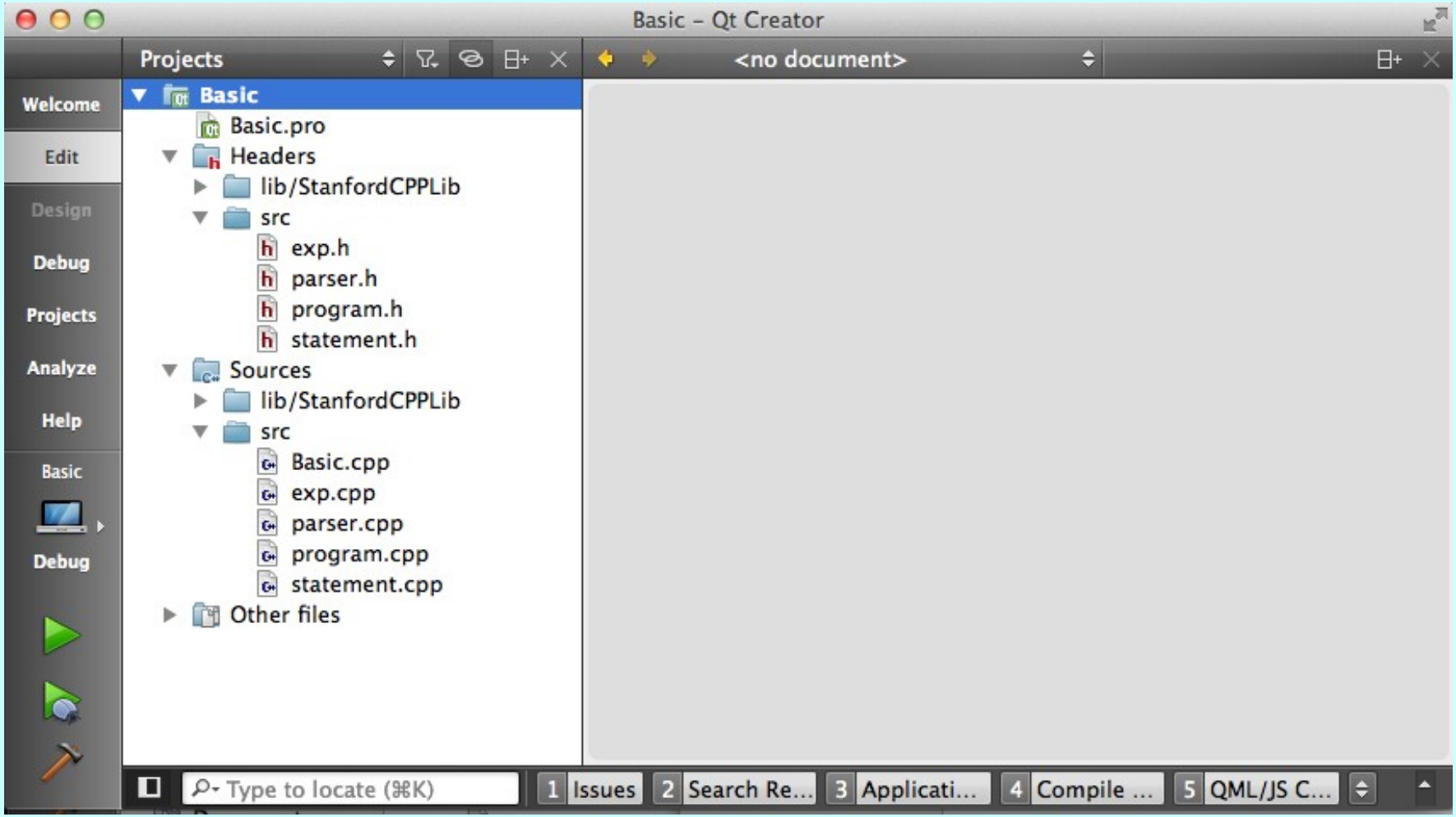
Please get going on this project as soon as possible.

Bill Gates

Exercise: Coding a BASIC Program

- On the second practice midterm, one of the problems concerned the ***hailstone sequence***. For any positive integer *n*, you compute the terms in the hailstone sequence by repeatedly executing the following steps:
  - If *n* is equal to 1, you've reached the end of the sequence and can stop.
  - If *n* is even, divide it by two.
  - If *n* is odd, multiply it by three and add one.
- Write a BASIC program that reads in an integer and prints out its hailstone sequence.

The Basic Starter Project

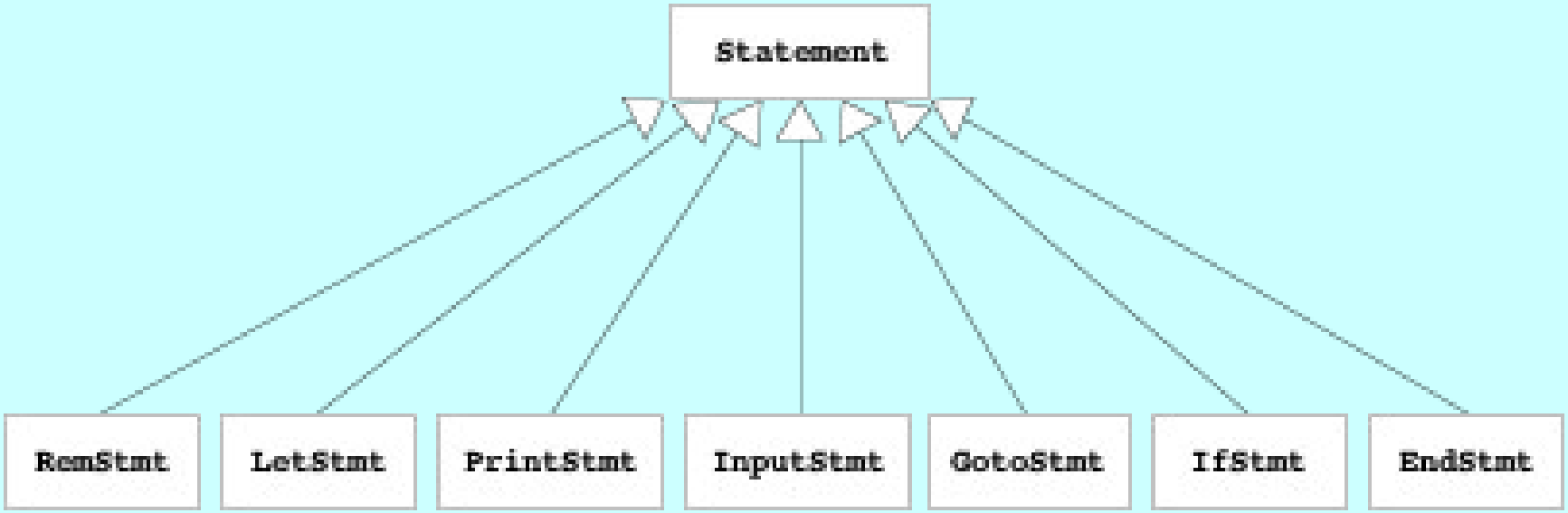


Modules in the Starter Folder

`Basic.cpp`	*You write this one, but it's short.*
`exp.h` **`exp.cpp`**	*You need to remove the* **`=`** *operator and add a few things to* **`EvaluationContext`**.
`parser.h` **`parser.cpp`**	*You need to remove the* **`=`** *operator.*
`program.h` **`program.cpp`**	*You're given the interface, but need to write the private section and the implementation.*
`statement.h` **`statement.cpp`**	*You're given the interface and need to supply the implementation.*

Your Primary Tasks

1. Figure out how the pieces of the program go together and what you need to do.
2. Code the **`Program`** class, keeping in mind what methods need to run in constant time.
3. Implement the **`Statement`** class hierarchy:



The End