CIS 343 - Structure of Programming Languages

Nathan Bowman

Slides by (slight modifications): Ira Woodring

Syntax and Semantics (Sebesta Text, Chapter 3)

SYNTAX

Written languages are sets of strings from some alphabet.

The strings are called sentences or statements.

Syntax tells us which strings are valid for the language.

SYNTAX

Syntax is the rules of the language.

For instance, in English we have a lot of rules:

- Start sentences with capital letters.
- End sentences with punctuation such as '.', '?', or '!'.
- Names are capitalized
- We use empty space between words.
- etc.

SYNTAX

If we have a string such as

aof;'(#;d,cla908a;sjam;aj;LJKha;e;)

we can say it is an invalid sentence in this language.

SYNTAX

Combinations of characters from the valid alphabet for the language can form **lexemes**, if those combinations express valid ideas in the language.

In English for example, we have lexemes such as "Go!", "apple", "it" and thousands others that express an idea.

Lexemes in a programming language may be numbers, operators, and special words.

SYNTAX

Lexemes are partitioned into groups. Each group is given a name or a **token**. For instance, variable names may be `a`, `student_name`, `car_two`. All of these lexemes are different, but they are all the same type of lexeme. We could label them with the token "identifiers".

SYNTAX

Sometimes a lexeme is the only element in a token. For instance we might have the operator +. Because of its specialized meaning we may just give it the token "addition_operator".

The number of possible valid sentences for a language are immense. Not feasible to provide language learners every possible sentence.

SYNTAX

In theory there are two ways we can define a language: by recognition or by generation.

SYNTAX

A Language Recognizer would be some device or software that takes as input strings of characters from a predefined alphabet and outputs whether the string is or is not valid.

This is what syntax analysis in a compiler does. Not good for learning a language though; must learn through trial and error.

SYNTAX

A Language Generator would be a device used to generate sentences in a language.

Seems to be only limitedly useful; we could use sample sentences to compare to our own though and begin to learn the language.

SYNTAX

To disseminate language to others, need a formal way to describe syntax (preferebly through generation rather than recognition).

Enter the Context-Free Grammar

SYNTAX

During the 1950s two researchers, in completely unrelated settings were working on this problem.

Noam Chomsky (a linguist) developed four classes of grammars for languages. Two of these *context-free*, and *regular* could be applied to programming languages, though these were not his interest.

SYNTAX

John Backus created a notation when designing ALGOL (later modified slightly by Peter Naur). This notation is called Backus-Naur Form (BNF).

A similar notation was used by Panini (an ancient Sanskrit linguist) to describe Sanskrit between the 4th and 6th centuries BCE.

This notation was very similar to Chomsky's ideas.

SYNTAX

BNF is a **metalanguage** - a language used to describe languages.

It uses abstractions for syntactic structures. For instance, an assignment statement definition may be given as:

```
<assign> -> <var> = <expression>
```

SYNTAX

What this describes (or really defines) for us is a rule for a valid sentence in a language.

Here we are saying that an assignment statement is defined as a variable followed by an equals sign followed by an expression.

SYNTAX

If we look closely at the statement again, we should make note of a few things:

```
<assign> -> <var> = <expression>
```

Every item we see in this sentence is a token of some sort. Some are called nonterminal symbols and some are terminal symbols.

SYNTAX

Nonterminals - abstractions. These may be broken down further to other tokens until they can be nothing more than some lexeme. May have two or more distinct definitions representing possible syntactic forms in the language.

Terminals - lexemes. These cannot be broken down any further.

Grammars - are the collections of rules for languages.

SYNTAX

When writing a grammar we may use the "|" symbol to separate multiple definitions of a nonterminal:

```
<if_stmt> -> if ( <logic_expr> ) <stmt>
<if_stmt> -> if ( <logic_expr> ) <stmt> else <stmt>
```

Or:

```
<if_stmt> -> if ( <logic_expr> ) <stmt>
           | -> if ( <logic_expr> ) <stmt> else <stmt>
```

SYNTAX

Lists are described in BNF recursively, i.e.:

```
<identifier_list> -> <identifier>
                   | <identifier>, <identifier_list>
```

SYNTAX

So we now have the ability to generate a language (a grammar is a language generator).

We begin with a start symbol and begin applying a sequence of rules. This is called creating a derivation.

For instance:

```
<program> -> begin <stmt_list> end
<stmt_list> -> <stmt>
            |  <stmt> ; <stmt_list>
<stmt> -> <var> = <expression>
<var> -> A | B | C
<expression> -> <var> + <var>
             |  <var> - <var>
             |  <var>
```

SYNTAX

This grammar describes VERY simple programs. These programs must begin with the keyword `begin` (a terminal symbol) and end with `end` (another terminal).

Allows for only assignment statements, possibly followed by a semicolon and another assignment.

Only three variable names are allowed, `A`, `B`, or `C`.

SYNTAX

Any sentence we can derive using these rules is a valid sentence in this language.

For instance, is

```
begin A = B + C ; B = C end
```

valid?

```
<program> => begin <stmt_list> end
          => begin <stmt> ; <stmt_list> end
          => begin <var> = <expression> ; <stmt_list> end
          => begin A = <expression> ; <stmt_list> end
          => begin A = <var> + <var> ; <stmt_list> end
          => begin A = B + <var> ; <stmt_list> end
          => begin A = B + C ; <stmt_list> end
          => begin A = B + C ; <stmt> end
          => begin A = B + C ; <var> = <expression> end
          => begin A = B + C ; B = <expression> end
          => begin A = B + C ; B = <var> end
          => begin A = B + C ; B = C end
```

Yes!

SYNTAX

Each line in the derivation is called a sentential form.

Here we performed leftmost derivations; we only replaced the nonterminal at the furthest left point in the sentence each time.

There are other ways!

Practice time!

Using this grammar:

```
<assign> -> <id> = <expr>
<id> -> A|B|C
<expr> -> <id> + <expr>
        | <id> * <expr>
        | ( <expr> )
        | <id>
```

Derive A = B * (A + C)

```
<assign> => <id> = <expr>
         => A = <expr>
         => A = <id> * <expr>
         => A = B * <expr>
         => A = B * ( <expr> )
         => A = B * ( <id> + <expr> )
         => A = B * ( A + <expr> )
         => A = B * ( A + <id> )
         => A = B * ( A + C )
```

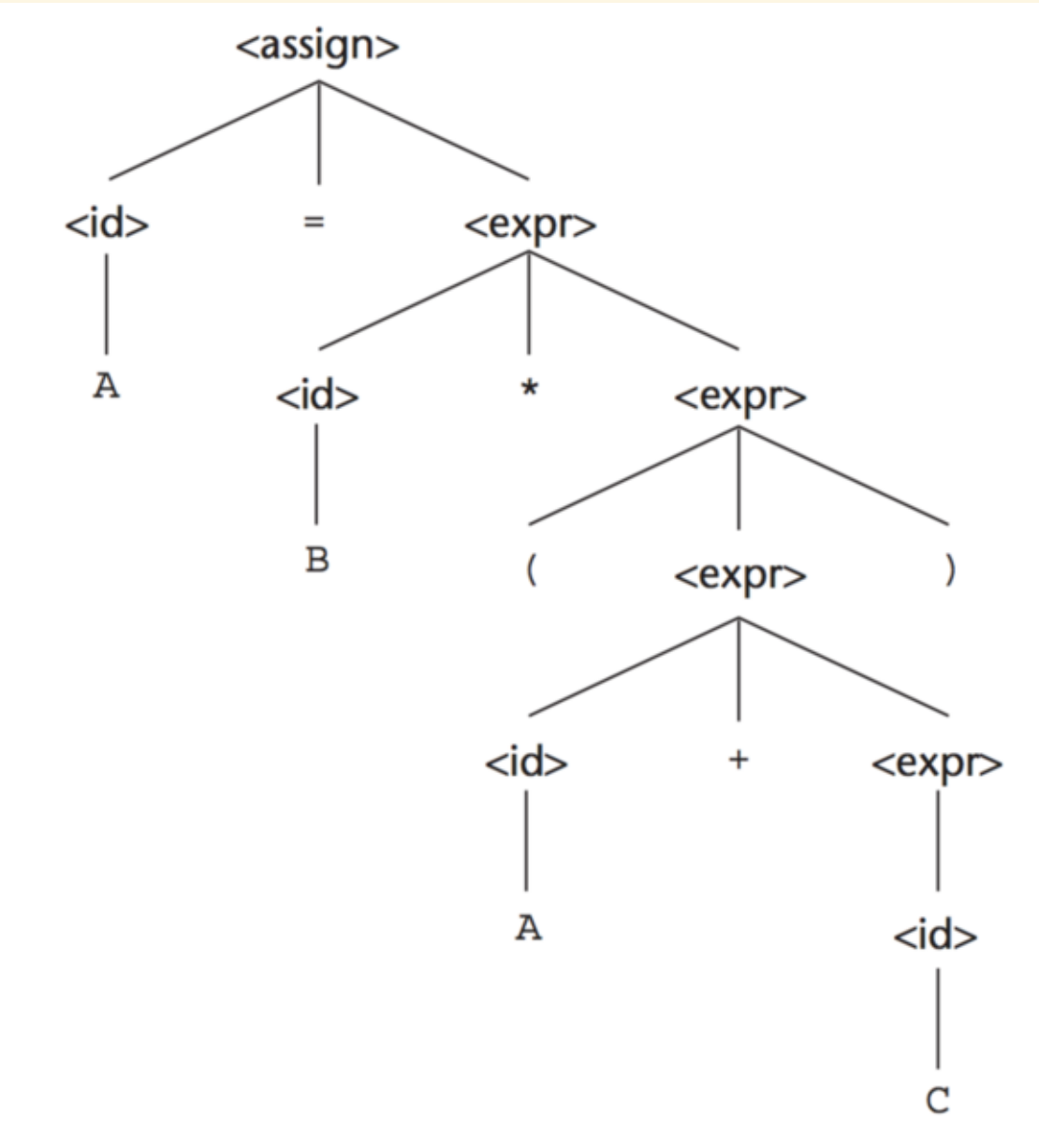
Syntax

Grammars do a great job of describing the hierarchy of the syntax. We can represent this hierarchy visually with parse trees.

The previous statement

```
A = B * ( A + C)
```

can be represented with the tree:


<assign>
<id>
=
<expr>
A
<id>
*
<expr>
B
(
<expr>
)
<id>
+
<expr>
A
<id>
C

Syntax

We must be VERY careful when designing a grammar. Consider this grammar:

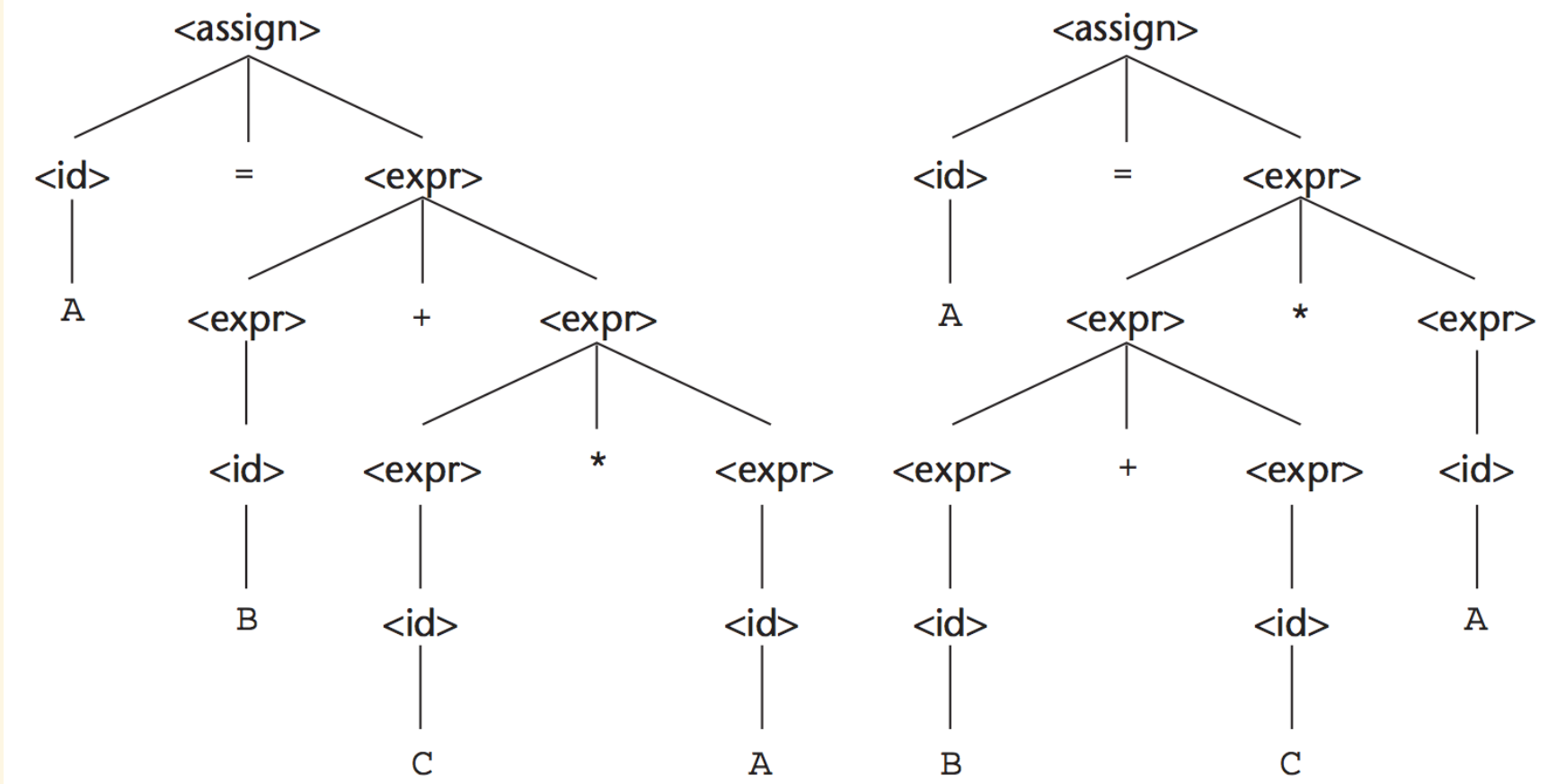
```
<assign> -> <id> = <expr>
<id> -> A | B | C
<expr> -> <expr> + <expr>
        | <expr> * <expr>
        | ( <expr> )
        | <id>
```

Syntax

Now, create a parse tree for this sentence:

```
A = B + C * A
```

THERE IS MORE THAN ONE POSSIBILITY!


<assign>
<id>
=
<expr>
A
<expr>
+
<expr>
<id>
<expr>
*
<expr>
B
<id>
<id>
C
A
<assign>
<id>
=
<expr>
A
<expr>
*
<expr>
<expr>
+
<expr>
<id>
<id>
<id>
A
B
C

Syntax

This grammar is ambiguous. Any grammar that has a sentential form for which there are two or more possible parse trees is ambiguous.

We don't want ambiguity in a grammar.

Syntax

What do we do if we have an ambiguous grammar?

- Provide some other non-grammatical information for the parser
- Rewrite the grammar

Syntax

Operator Precedence is one way of providing non-grammatical information.

We assign different levels of importance to a token so that it is evaluated first if it is found at the same level as another token.

With the previous grammar, if our parser knew the rules of mathematics (which are NOT grammar rules), and could apply those we wouldn't have a problem.

Syntax

Note that in our first grammar:

```
<assign> -> <id> = <expr>
<id> -> A|B|C
<expr> -> <id> + <expr>
        | <id> * <expr>
        | ( <expr> )
        | <id>
```

We still need operator precedence. This one is unambiguous but does not follow math rules:

Syntax

```
A + B * C        // Parse tree will evaluate A + B first
```

May not be mathematically correct.

```
A * B + C        // Parse tree hierarchy correct here
```

Could be.

Syntax

We could rewrite the original grammar as well. We could add some rules and new nonterminals:

Syntax

This grammar will produce the same language, but without ambiguity:

```
<assign> -> <id> = <expr>
<id> -> A | B | C
<expr> -> <expr> + <term>
        | <term>
<term> -> <term> * <factor>
        | <factor>
<factor> -> ( <expr> )
        | <id>
```

Syntax

Now generate

```
A = B + C * A
```

```
<assign> => <id> = <expr>
         =>A = <expr>
         =>A = <expr> + <term>
         =>A = <term> + <term>
         =>A = <factor> + <term>
         =>A = <id> + <term>
         =>A = B + <term>
         =>A = B + <term> * <factor>
         => A = B + <factor> * <factor>
         =>A = B + <id> * <factor>
         =>A = B + C * <factor>
         =>A = B + C * <id>
         =>A = B + C * A
```

Syntax

When we have operators with the same precedence level, we must have a semantic rule to decide which should come first. This is the concept of *associativity*.

Syntax

At times we may have a grammar rule that includes the left-hand side (LHS) at the beginning of the right-hand side (RHS). For example:

```
<term> -> <term> * <factor>
        | <factor>
```

This rule is considered "left recursive". This implies left associativity. We may use this technique to specify that addition or multiplication are left-associative (we perform the calculation left to right).

Syntax

We may also have "right recursive" rules. These may be useful, for instance, when we have a scenario like exponentiation. Exponents should be calculated right to left (2^2^3 = 2^(2^3) = 2^(8) = 256).

```
<factor> -> <exp> ** <factor>
          | <exp>
<exp> -> ( <expr> )
        | id
```

Syntax

There are various extensions to BNF, many of which are (somewhat confusingly) called Extended BNF (EBNF)

These extensions do not change *what* we can express in BNF, just how easy some things are to express

Most extensions have three common features

Syntax

Extension 1: **Optional part**

Denoted with [ ]

```
<if_stmt> -> if (<expression>) <statement> [else <statement>]
```

Syntax

Extension 2: **Repeat 0 or more times**

Denoted with { }

```
<ident_list> -> <identifier> {, <identifier>}
```

Syntax

Extension 3: **Multiple-choice options**

Denoted with ( ) and |

```
<term> -> <term> ( * | / | % ) <factor>
```

Static Semantics -- Attribute Grammars

Attribute grammars are an extension to a Context-Free Grammar, but not in the sense of EBNF -- attribute grammars can express concepts that context-free grammars cannot

Allows some language rules to be described more easily.

Consider for instance, type compatibility rules in Java.

Static Semantics -- Attribute Grammars

A float can't be assigned to an int.

But an int can be assigned to a float.

We can describe this with BNF, but it adds more terminal symbols and rules. The more of these we add to the language, the larger the grammar grows and the syntax analyzer grows as well (not good!).

Static Semantics -- Attribute Grammars

Additionally, consider the situation where a language requires a variable to be declared before it is used. According to our text, this has been proven to be impossible to describe with BNF.

These problems are called *static semantics*.

They are only partially related to a program while it is running; they are more concerned with compile time constraints (hence the **static**).

Static Semantics -- Attribute Grammars

Attribute Grammars are CFGs which have attributes, attribute computation functions, and predicate functions.

They may, for instance, be used to specify that a language that uses a `BEGIN PROCEDURE_NAME` rule is ended by an `END` and the same `PROCEDURE_NAME`, or to easily address the situations mentioned in the past slides.

Static Semantics -- Attribute Grammars

We won't be discussing attribute grammars further in this class; though you should be reading the sections that include them. For now, they are included in this section to help you understand more fully what can and cannot be described with BNF forms.

Semantics - The Meaning of Programs

Where syntax refers to the form of a program, **semantics** refers to the meaning.

Turns out describing syntax is easy. Semantics, not so much.

(To distinguish from static semantics, we could call this "dynamic semantics", but we shorten it to simply "semantics")

Semantics - The Meaning of Programs

People using a language need to know what language statements will do.

Unfortunately we often determine the semantics from manuals written in a human language, and human languages are imprecise and incomplete (as we saw when we learned about specific words in certain languages that English doesn't have!).

Semantics - The Meaning of Programs

A few methods have been designed to help formally denote semantics.

- operational semantics
- denotational semantics
- axiomatic semantics

Semantics - The Meaning of Programs

Operational Semantics attempts to describe the meaning of a statement or program by specifying the effects of running it on a machine.

Problem is, too many changes (and changes too small) usually in machine state.

Instead, create an idealized virtual machine with simple machine language -- an interpreter

Semantics - The Meaning of Programs

One way to go: run the program and see what happens

If you need only understand individual statements, rather than program as a whole, not actually necessary to build the interpreter and run

Examine the expression in idealized machine code yourself

```
C statement                          Meaning

for (expr1 ; expr2 ; expr3) {            expr1;
    ...                             loop: if expr2 == 0 goto out
}                                        ...
                                         expr3;
                                         goto loop
                                    out: ...
```

Semantics - The Meaning of Programs

Rewriting higher-level construct in (made up) lower-level language

Important to choose the right balance for lower-level language -- do not want to use actual machine code because it is not as clear

Least formal of three methods for studying semantics discussed here

Semantics - The Meaning of Programs

Denotational Semantics maps language entities to mathematical objects through functions

Okay...

Semantics - The Meaning of Programs

Similar to operational semantics in that we map code to something else and then look at that something else for the meaning

Think of it as creating functions that take in syntax and output meaning (semantics)

For instance, if we had the grammar

```
<bin_num> -> '0'
           | '1'
           | <bin_num> '0'
           | <bin_num> '1'
```

Semantics - The Meaning of Programs

We could map this to the objects

```
Mbin('0') = 0
Mbin('1') = 1
Mbin(<bin_num> '0') = 2 * Mbin(<bin_num>)
Mbin(<bin_num> '1') = 2 * Mbin(<bin_num>) + 1
```

With this we can construct the meaning of this entity, where we decided that "meaning" is the equivalent decimal number

Using this function, the meaning of

```
1011
```

is

```
Mbin('1011') = 2 * (Mbin(101)) + 1
     = 2 * (2 * (Mbin 10) + 1) + 1
     = 2 * (2 * (2 * (Mbin 1)) + 1) + 1
     = 2 * (2 * (2 * 1) + 1) + 1
     = 2 * (2 * (2) + 1) + 1
     = 2 * (4 + 1) + 1
     = 2 * (5) + 1
     = 10 + 1
     = 11
```

Semantics - The Meaning of Programs

We don't just write these "meaning functions" for constants like `1011`

Programming *constructs* get their own meaning functions

Meaning function for a `while` loop could be denoted

```
Ml (while B do L, s)
```

Apply functions to determine effect on state `s`

Semantics - The Meaning of Programs

According to your book, denotational semantics are

- highly rigorous
- an excellent way to describe languages concisely
- a useful tool for language designers
- not particularly useful to programmers

Semantics - The Meaning of Programs

Axiomatic Semantics is the most abstract branch of formal semantics. This field is concerned with what can be proven about a program, such as a proof of correctness.

Axiomatic Semantics uses logical expressions called assertions to describe constraints on variables.

Each expression has preconditions and postconditions.

For instance:

```
{x > 0} sum = 2 * x + 1 {sum > 1}
```

Semantics - The Meaning of Programs

Validation of a program using Axiomatic Semantics will be a proof

Start with postconditions and deduce preconditions

If input conditions are what is expected, program is *provably correct*

```
{x = A AND y = B}
t = x;
x = y;
y = t;
{x = B AND y = A}
```

Semantics - The Meaning of Programs

Studying axiomatic semantics:

- is useful for research into provably correct programs
- can be a useful way to reason about your programs
- is of limited use in understanding a programming language