CSS131:Theory of Programming Languages

Chapter 1: Syntax

Asst. Prof. Peerasak Intarapaiboon

Contents

1.1 Grammars
 1.1.1 Backus-Naur Form
 1.1.2 Derivations
 1.1.3 Parse Trees
 1.1.4 Associativity and Precedence
 1.1.5 Ambiguous Grammars
1.2 Extended BNF
1.3 Compilers and Interpreters
1.4 Linking Syntax and Semantics
 1.4.1 Abstract Syntax
 1.4.2 Abstract Syntax Trees

Sentence	Vocab	Grammar
The chased cat the dog.		
Today I see the dog.		
Very happy the student is.		
Ich sehe den Hund.		
the cat chase a dog.		

Thinking about Syntax

The *syntax* of a programming language is a precise description of all its grammatically correct programs.

Precise syntax was first used with Algol 60, and has been used ever since.

Three levels:

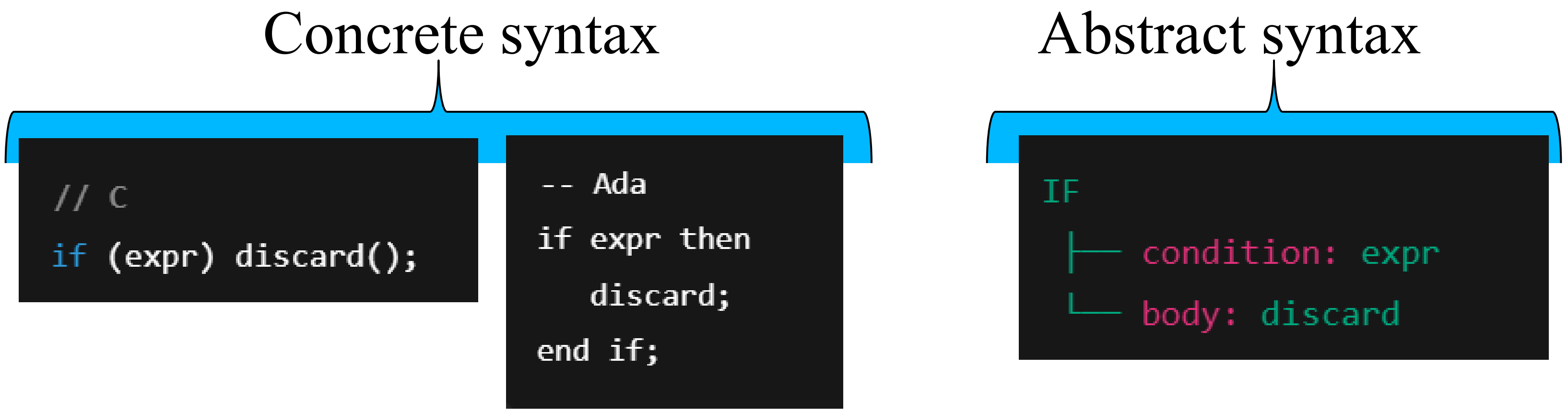
- ***Lexical syntax***
- ***Concrete syntax***
- ***Abstract syntax***

Levels of Syntax

Lexical syntax = all the basic symbols of the language (names, values, operators, etc.)

Concrete syntax = rules for writing expressions, statements and programs.

Abstract syntax = internal representation of the program, favoring content over form. E.g.,


Concrete syntax

```
// C
if (expr) discard();
```

```
-- Ada
if expr then
   discard;
end if;
```

Abstract syntax

```
IF
├── condition: expr
└── body: discard
```

Grammars

A *metalanguage* is a language used to define other languages.

A *grammar* is a metalanguage used to define the syntax of a language.

Our interest: using grammars to define the syntax of a programming language.

Backus-Naur Form (BNF)

- Stylized version of a context-free grammar (cf. Chomsky hierarchy)
- Sometimes called Backus Normal Form
- First used to define syntax of Algol 60
- Now used to define syntax of most major languages

BNF Grammar

Set of *productions*: P

terminal symbols: T

nonterminal symbols: N

start symbol: S

A *production* has the form

$A \rightarrow w$

where $A \in N$ and $w \in (T \cup N)^*$

Example: Binary Digits

Consider the grammar:

$$binaryDigit \rightarrow 0$$
$$binaryDigit \rightarrow 1$$

or equivalently:

$$binaryDigit \rightarrow 0 \mid 1$$

Here, | is a metacharacter that separates alternatives.

Derivations

Consider the grammar:

$$Integer \rightarrow Digit \mid Integer\ Digit$$

$$Digit \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

We can *derive* any unsigned integer, like 352, from this grammar.

Derivation of 352 as an *Integer*

A 6-step process, starting with:

Integer

Derivation of 352 (step 1)

Use a grammar rule to enable each step:

$$Integer \Rightarrow Integer\ Digit$$

Derivation of 352 (steps 1-2)

Replace a nonterminal by a right-hand side of one of its rules:

$$\begin{aligned} Integer &\Rightarrow Integer\ Digit \\ &\Rightarrow Integer\ 2 \end{aligned}$$

Derivation of 352 (steps 1-3)

Each step follows from the one before it.

$$\begin{aligned} Integer &\Rightarrow Integer\ Digit \\ &\Rightarrow Integer\ 2 \\ &\Rightarrow Integer\ Digit\ 2 \end{aligned}$$

Derivation of 352 (steps 1-4)

$$\begin{aligned} Integer &\Rightarrow Integer\ Digit \\ &\Rightarrow Integer\ 2 \\ &\Rightarrow Integer\ Digit\ 2 \\ &\Rightarrow Integer\ 5\ 2 \end{aligned}$$

Derivation of 352 (steps 1-5)

$$\begin{aligned}
Integer &\Rightarrow Integer\ Digit \\
&\Rightarrow Integer\ 2 \\
&\Rightarrow Integer\ Digit\ 2 \\
&\Rightarrow Integer\ 5\ 2 \\
&\Rightarrow Digit\ 5\ 2
\end{aligned}$$

Derivation of 352 (steps 1-6)

You know you're finished when there are only terminal symbols remaining.

$$\begin{aligned}
\mathit{Integer} &\Rightarrow \mathit{Integer}\ \mathit{Digit} \\
&\Rightarrow \mathit{Integer}\ 2 \\
&\Rightarrow \mathit{Integer}\ \mathit{Digit}\ 2 \\
&\Rightarrow \mathit{Integer}\ 5\ 2 \\
&\Rightarrow \mathit{Digit}\ 5\ 2 \\
&\Rightarrow 3\ 5\ 2
\end{aligned}$$

A Different Derivation of 352

$$\begin{aligned} \mathit{Integer} &\Rightarrow \mathit{Integer}\ \mathit{Digit} \\ &\Rightarrow \mathit{Integer}\ \mathit{Digit}\ \mathit{Digit} \\ &\Rightarrow \mathit{Digit}\ \mathit{Digit}\ \mathit{Digit} \\ &\Rightarrow 3\ \mathit{Digit}\ \mathit{Digit} \\ &\Rightarrow 3\ 5\ \mathit{Digit} \\ &\Rightarrow 3\ 5\ 2 \end{aligned}$$

This is called a *leftmost derivation*, since at each step the leftmost nonterminal is replaced.

(The first one was a *rightmost derivation*.)

Notation for Derivations

$Integer \Rightarrow^* 352$

Means that 352 can be derived in a finite number of steps using the grammar for *Integer*.

$352 \in L(G)$

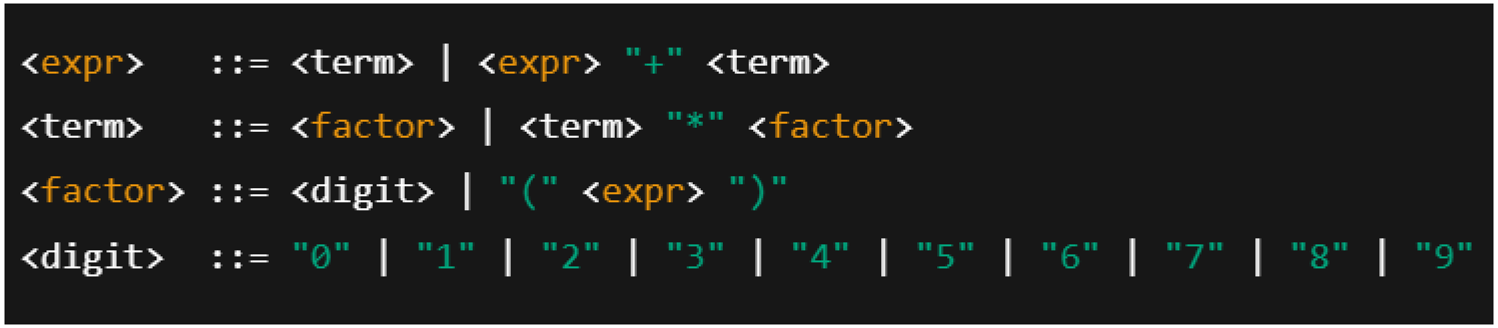
Means that 352 is a member of the language defined by grammar *G*.

$L(G) = \{ \omega \in T^* \mid Integer \Rightarrow^* \omega \}$

Means that the language defined by grammar *G* is the set of all symbol strings ω that can be derived as an *Integer*.

Practice

กำหนด Grammar ดังนี้



```
<expr>   ::= <term> | <expr> "+" <term>
<term>   ::= <factor> | <term> "*" <factor>
<factor> ::= <digit> | "(" <expr> ")"
<digit>  ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

พิจารณาว่า String ต่อไปนี้ สามารถสร้างจาก Grammar นี้ได้หรือ ไม่

A. 2+3*4

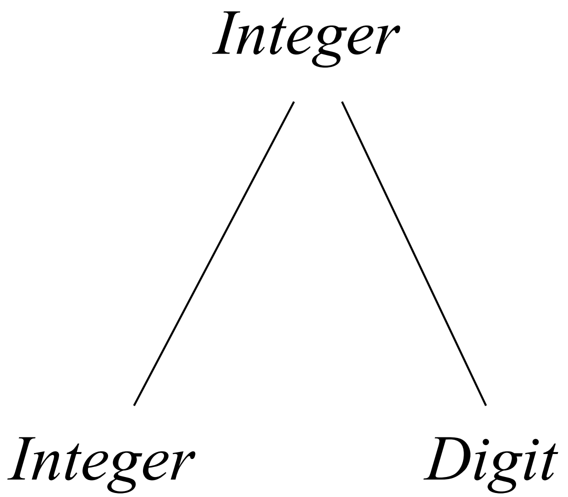
B. 1 + (2)

Parse Trees

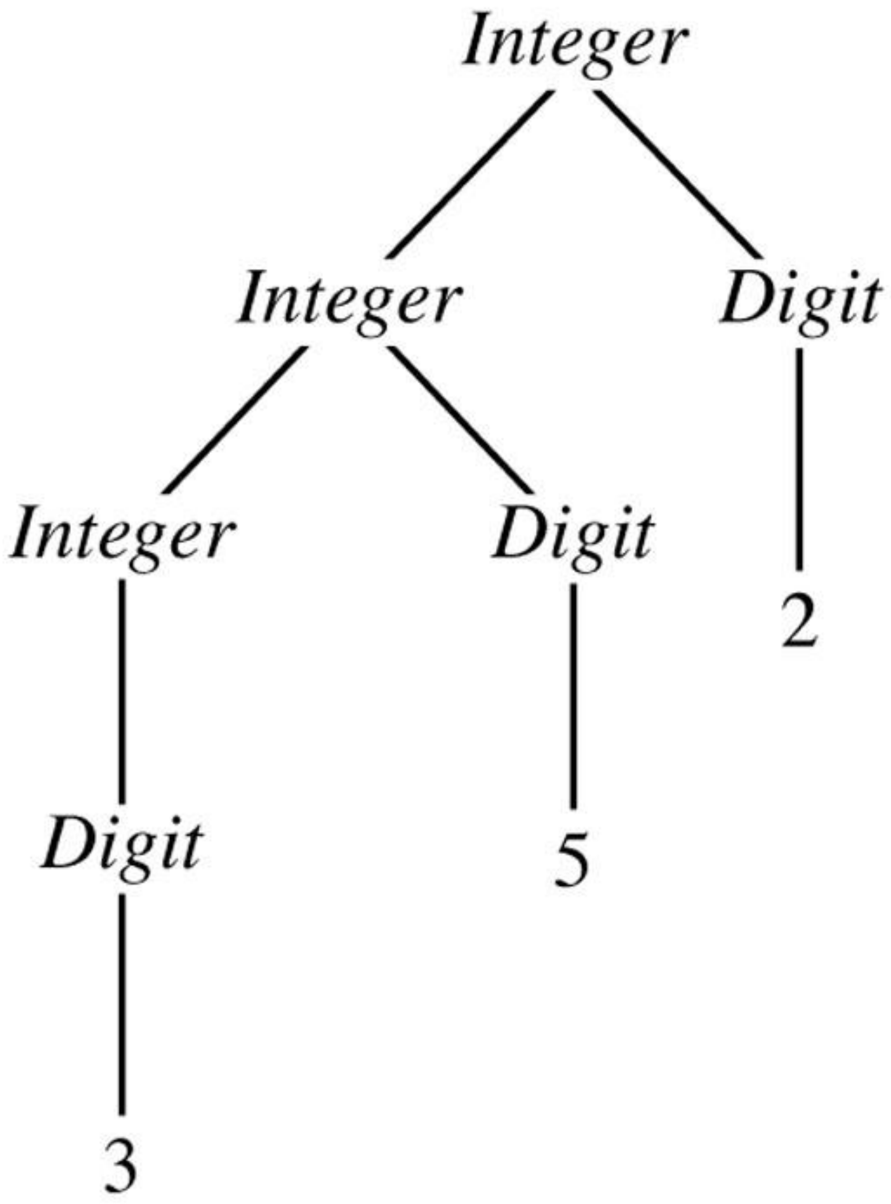
A *parse tree* is a graphical representation of a derivation.

- *Each internal node of the tree corresponds to a step in the derivation.*
- *Each child of a node represents a right-hand side of a production.*
- *Each leaf node represents a symbol of the derived string, reading from left to right.*

E.g., The step $\mathit{Integer} \Rightarrow \mathit{Integer}\ \mathit{Digit}$ appears in the parse tree as:



Parse Tree for 352 as an *Integer*



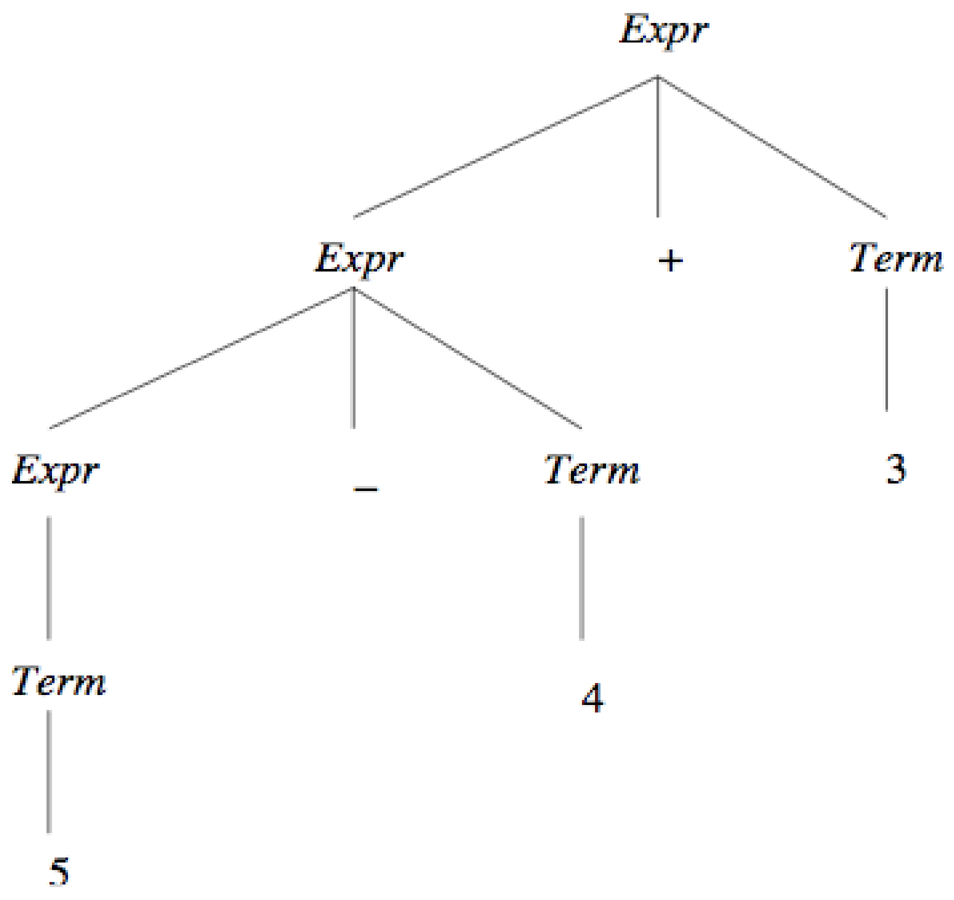
Arithmetic Expression Grammar

The following grammar defines the language of arithmetic expressions with 1-digit integers, addition, and subtraction.

$$Expr \rightarrow Expr + Term \mid Expr - Term \mid Term$$

$$Term \rightarrow 0 \mid \ldots \mid 9 \mid (\; Expr \;)$$

Parse of the String 5-4+3



Associativity and Precedence

A grammar can be used to define associativity and precedence among the operators in an expression.

E.g., + and - are left-associative operators in mathematics;

** and / have higher precedence than + and - .*

Consider the more interesting grammar G_1:

Expr -> Expr + Term | Expr – Term | Term

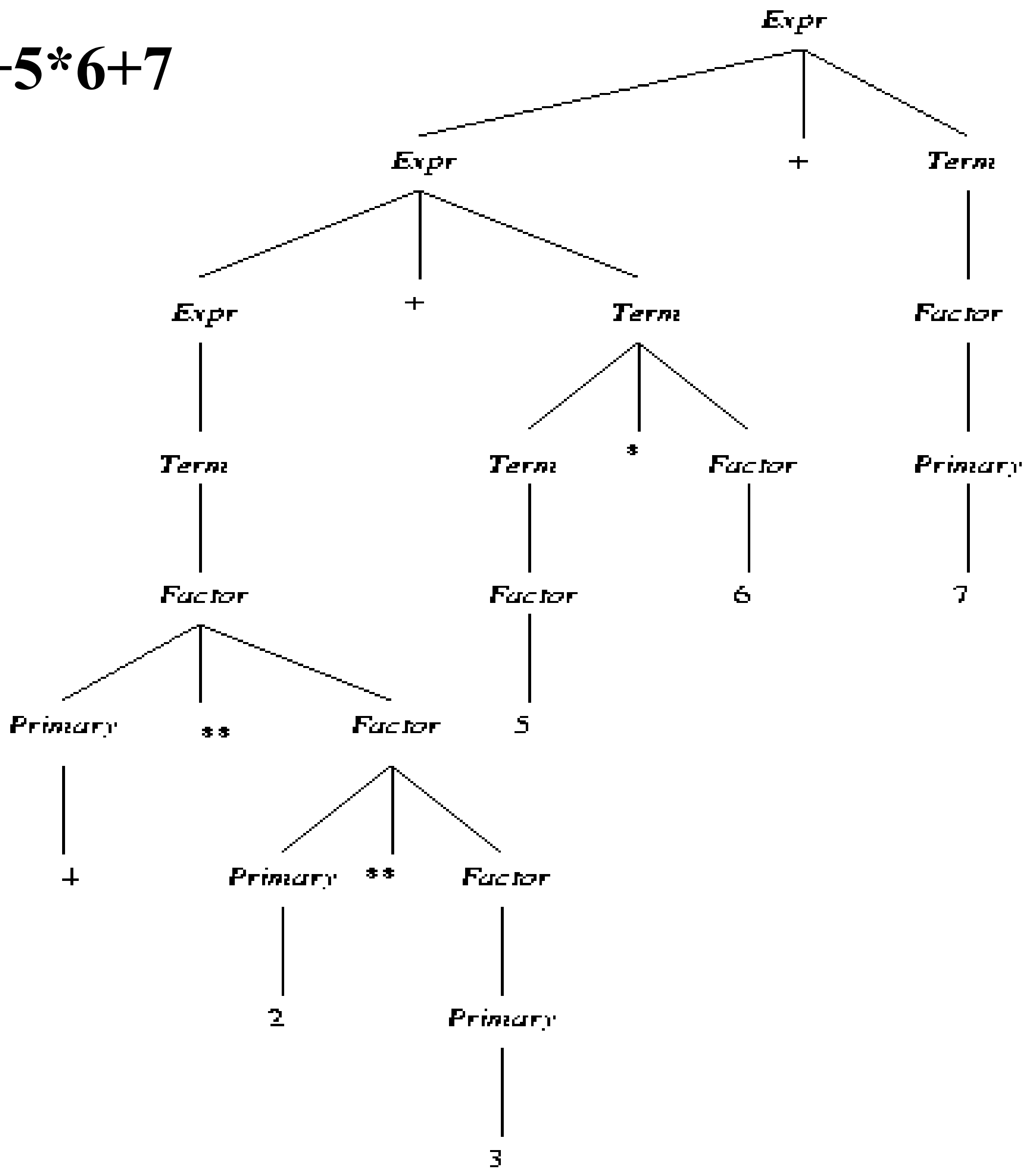
*Term -> Term * Factor | Term / Factor |*

Term % Factor | Factor

*Factor -> Primary ** Factor | Primary*

Primary -> 0 | … | 9 | (Expr)

Parse of 4**2**3+5*6+7 for Grammar G_1



Associativity and Precedence for Grammar G_1

Precedence	Associativity	Operators
3	right	**
2	left	* / %
1	left	+ -

Note: These relationships are shown by the structure of the parse tree: highest precedence at the bottom, and left-associativity on the left at each level.

Ambiguous Grammars

A grammar is *ambiguous* if one of its strings has two or more diffferent parse trees.

E.g., Grammar G$_1$ *above is unambiguous.*

C, C++, and Java have a large number of

- *operators and*
- *precedence levels*

Instead of using a large grammar, we can:

- *Write a smaller ambiguous grammar, and*
- *Give separate precedence and associativity (e.g., Table 2.1)*

An Ambiguous Expression Grammar G_2

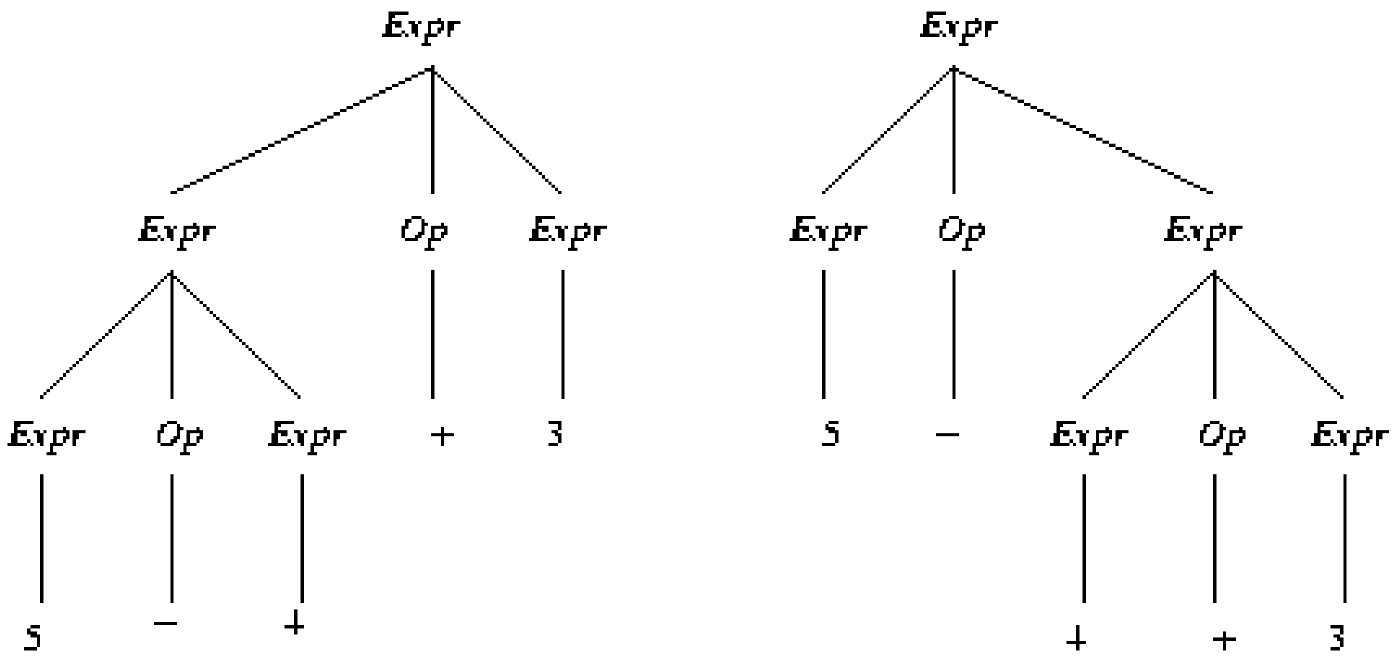
Expr -> Expr Op Expr | (*Expr*) | *Integer*

Op -> + | - | * | / | % | **

Notes:

- G_2 is equivalent to G_1. I.e., its language is the same.
- G_2 has fewer productions and nonterminals than G_1.
- However, G_2 is ambiguous.

Ambiguous Parse of 5-4+3 Using Grammar G_2



(a)

(b)

The Dangling Else

IfStatement -> if (*Expression*) *Statement* |

    if (*Expression*) *Statement* else *Statement*

Statement -> *Assignment* | *IfStatement* | *Block*

Block -> { *Statements* }

Statements -> *Statements Statement* | *Statement*

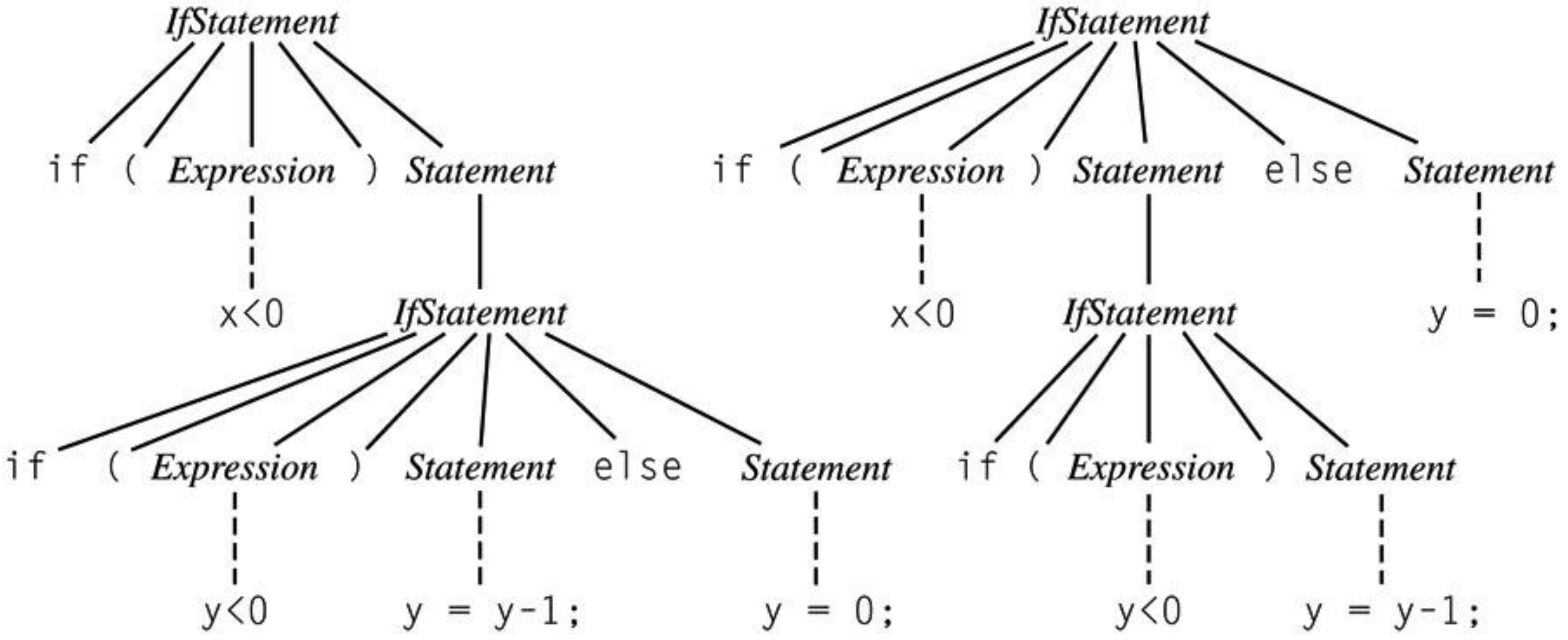
Example

With which ‘if’ does the following ‘else’ associate

```
        if (x < 0)
                if (y < 0)  y = y  - 1;
                else y = 0;
```

Answer: *either one!*

The *Dangling Else* Ambiguity



Solving the dangling else ambiguity

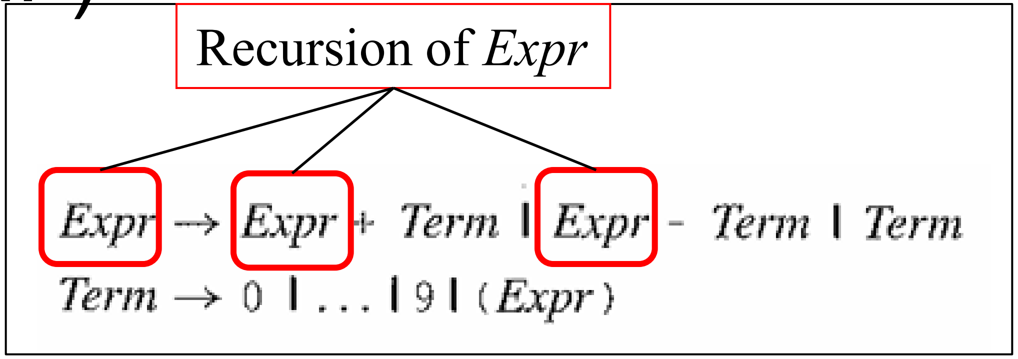
1. Algol 60, C, C++: associate each else with closest if; use {} or begin…end to override.
2. Algol 68, Modula, Ada: use explicit delimiter to end every conditional (e.g., if…fi)
3. Java: rewrite the grammar to limit what can appear in a conditional:

IfThenStatement -> if (*Expression*) *Statement*

IfThenElseStatement -> if (*Expression*) *StatementNoShortIf* else *Statement*

The category *StatementNoShortIf* includes all except *IfThenStatement*.

Extended BNF (EBNF)


Recursion of *Expr*
Expr → *Expr* + *Term* | *Expr* - *Term* | *Term*
Term → 0 | . . . | 9 | (*Expr*)

BNF:

- *recursion for iteration*
- *nonterminals for grouping*

EBNF: additional metacharacters

- **{ }** for a series of zero or more
- **( )** for a list, must pick one
- **[ ]** for an optional list; pick none or one

EBNF Examples

Expression is a list of one or more *Terms* separated by operators + and -

Expression -> *Term* **{ (+ | -)** *Term* **}**

IfStatement -> if (*Expression*) *Statement* **[** else *Statement* **]**

C-style EBNF lists alternatives vertically and uses $_{opt}$ *to signify optional parts. E.g.,*

IfStatement:

if (*Expression*) *Statement* *ElsePart*$_{opt}$

ElsePart:

else *Statement*

EBNF to BNF

1. $A \rightarrow a\{x\}b \equiv ax^{*}b = ab \mid a\,xb \mid a\,xxb \mid axxxb \mid \ldots$

   Replace $\{x\}$ by R and add rule $R \rightarrow \epsilon \mid xR$

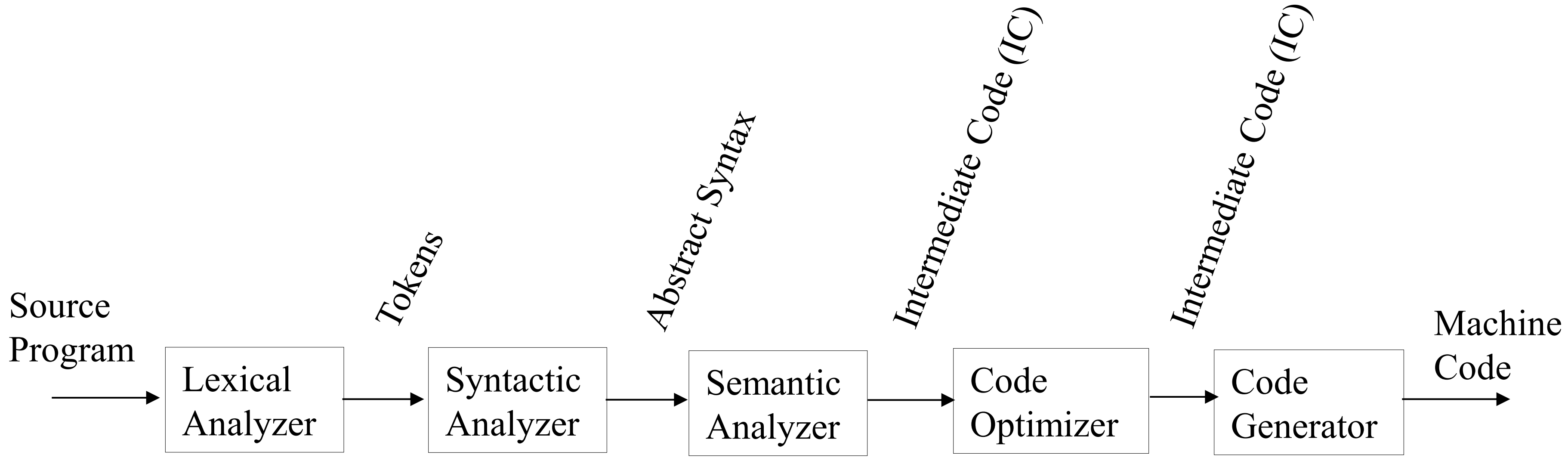
   $A \rightarrow aRb$

   $R \rightarrow \epsilon|xR$

2. $\mathrm{A} \rightarrow a(+|-)b$

3. $\mathrm{A} \rightarrow a[x]b$

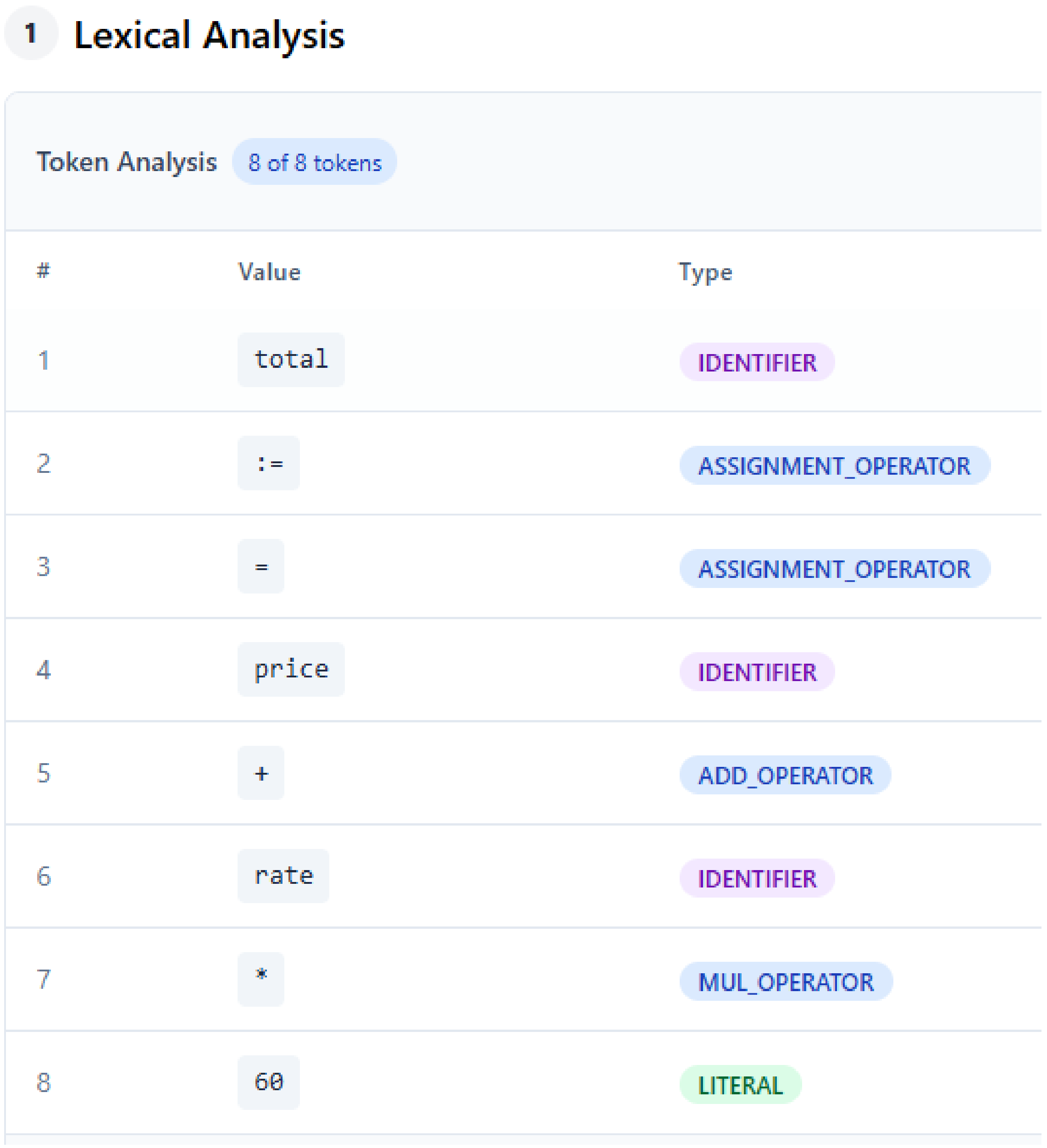
Compilers and Interpreters



Lexer

- Input: characters
- Output: tokens
- Separate:
  - *Speed: 75% of time for non-optimizing*
  - *Simpler design*
  - *Character sets*

total := price + rate * 60



1 Lexical Analysis

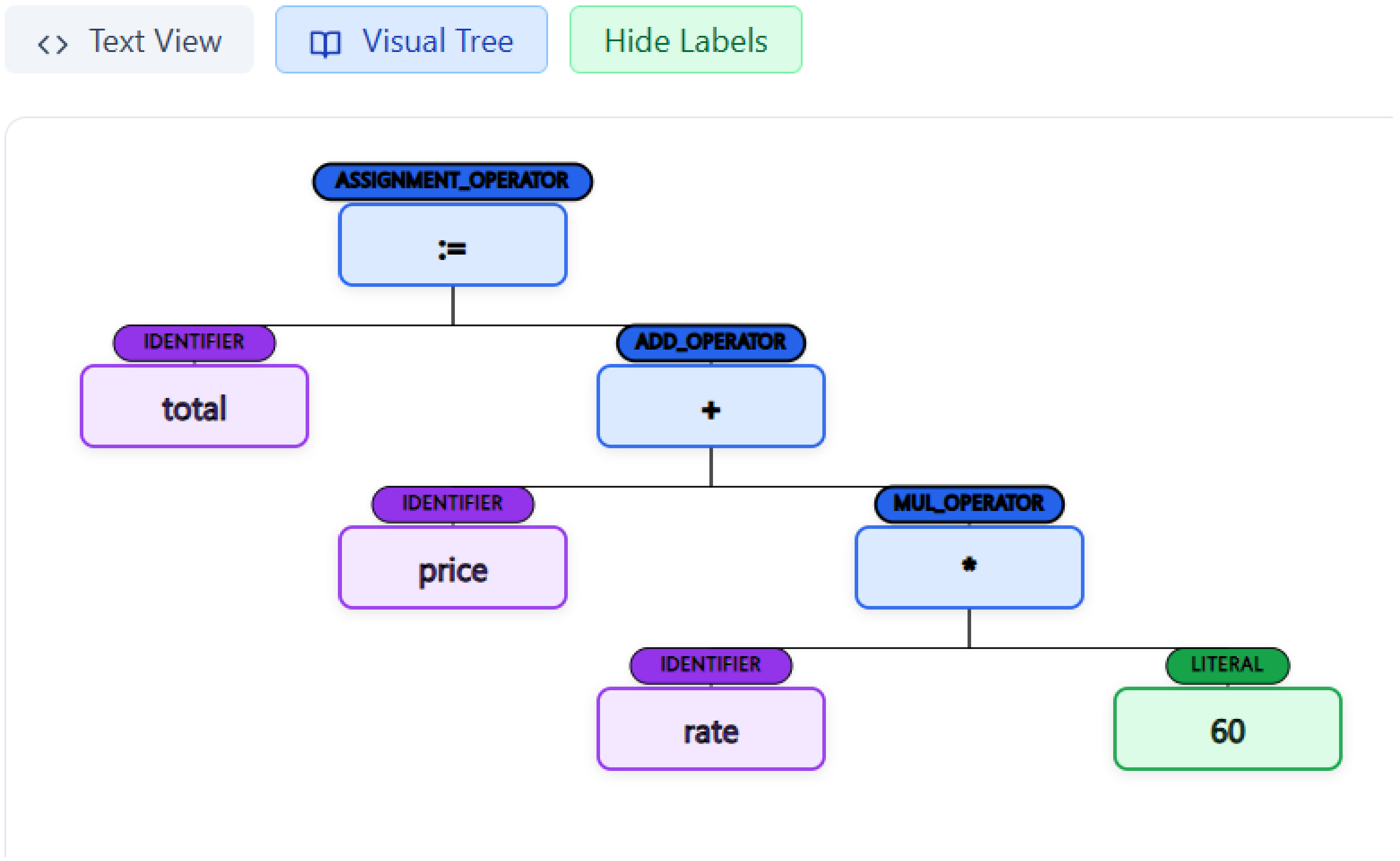
Token Analysis 8 of 8 tokens

#	Value	Type
1	total	IDENTIFIER
2	:=	ASSIGNMENT_OPERATOR
3	=	ASSIGNMENT_OPERATOR
4	price	IDENTIFIER
5	+	ADD_OPERATOR
6	rate	IDENTIFIER
7	*	MUL_OPERATOR
8	60	LITERAL

Parser

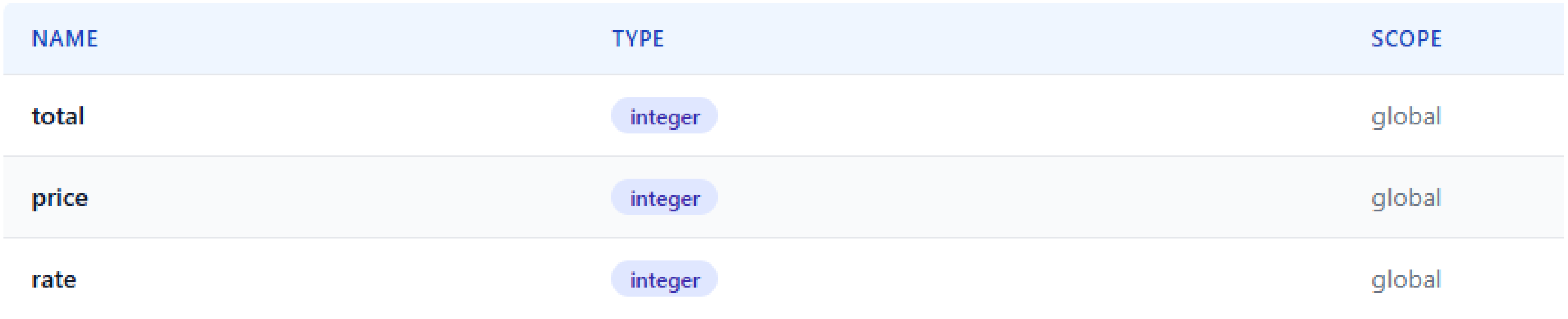
- Based on BNF/EBNF grammar
- Input: tokens
- Output: abstract syntax tree (parse tree)
- Abstract syntax: parse tree with punctuation, many nonterminals discarded

2 Syntax Analysis



Semantic Analysis

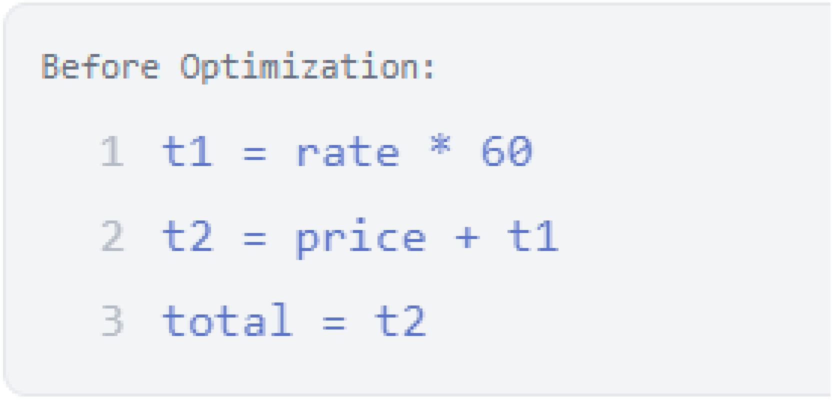
- Check that all identifiers are declared
- Perform type checking
- Insert implied conversion operators (i.e., make them explicit)



NAME	TYPE	SCOPE
total	integer	global
price	integer	global
rate	integer	global

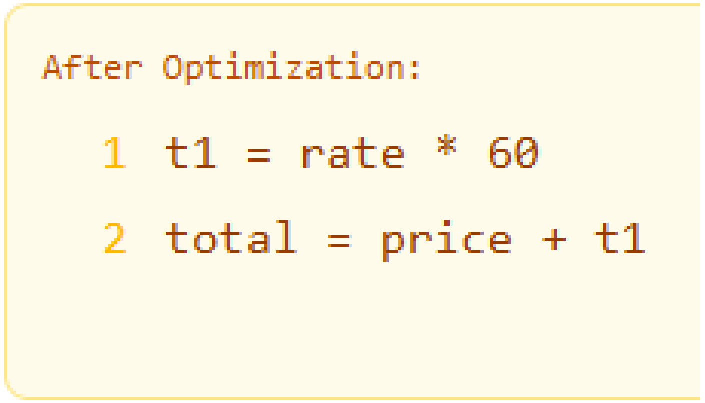
Code Optimization

- Evaluate constant expressions at compile-time
- Reorder code to improve cache performance
- Eliminate common subexpressions
- Eliminate unnecessary code



Before Optimization:

```
t1 = rate * 60
t2 = price + t1
total = t2
```

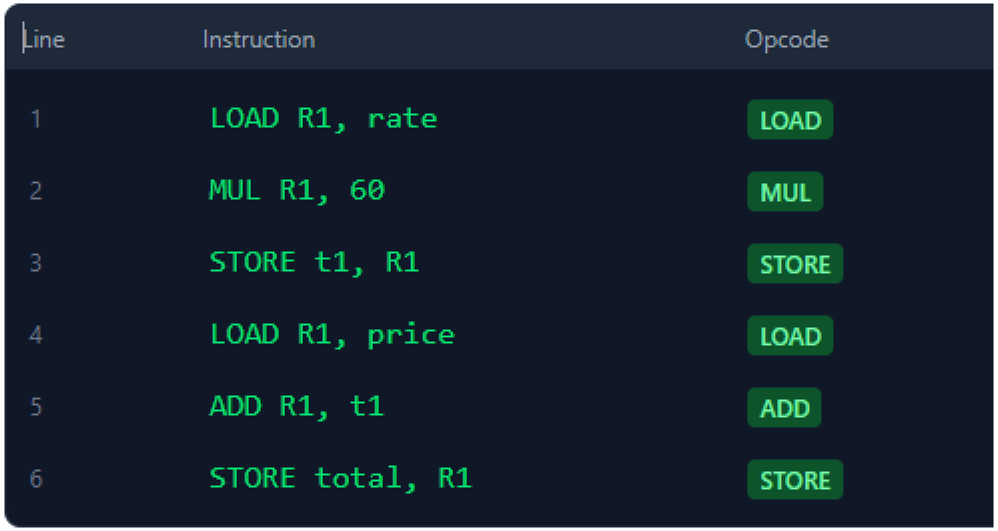


After Optimization:

```
t1 = rate * 60
total = price + t1
```

Code Generation

- Output: machine code
- Instruction selection
- Register management
- Peephole optimization



Line	Instruction	Opcode
1	`LOAD R1, rate`	LOAD
2	`MUL R1, 60`	MUL
3	`STORE t1, R1`	STORE
4	`LOAD R1, price`	LOAD
5	`ADD R1, t1`	ADD
6	`STORE total, R1`	STORE

Interpreter

Input:

- *Mixed: intermediate code*
- *Pure: stream of ASCII characters*

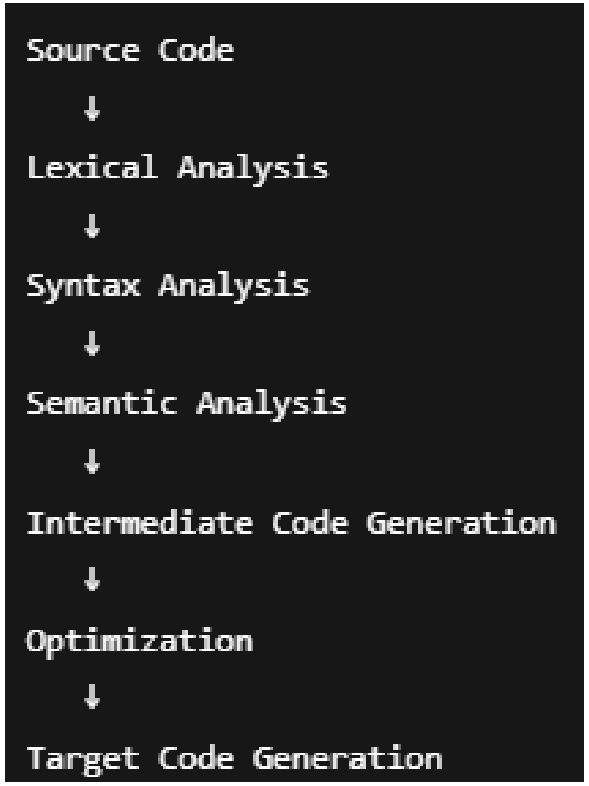
Mixed interpreters

- *Java, Perl, Python, Haskell, Scheme*

Pure interpreters:

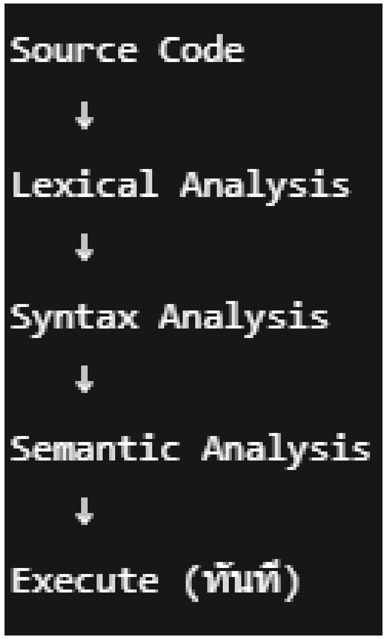
- *most Basics, shell commands*

Compiler



```
Source Code
  ↓
Lexical Analysis
  ↓
Syntax Analysis
  ↓
Semantic Analysis
  ↓
Intermediate Code Generation
  ↓
Optimization
  ↓
Target Code Generation
```

Interpreter



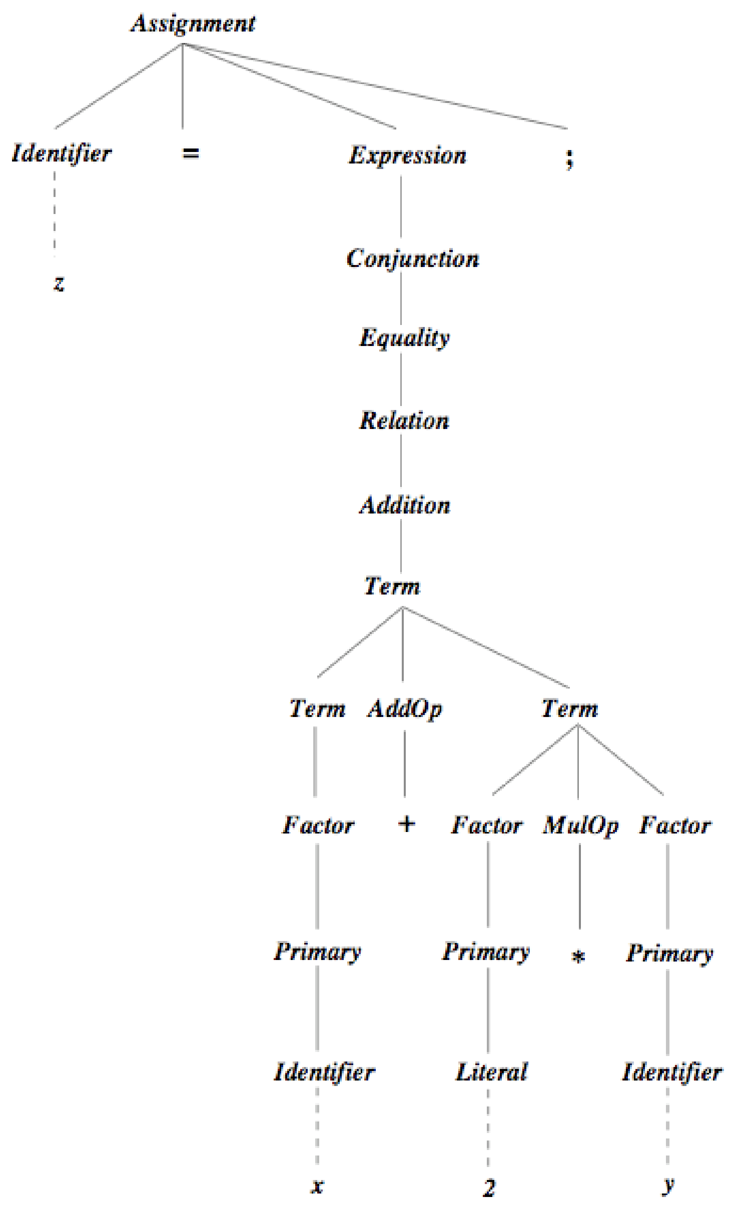
```
Source Code
  ↓
Lexical Analysis
  ↓
Syntax Analysis
  ↓
Semantic Analysis
  ↓
Execute (ทันที)
```

Linking Syntax and Semantics

Output: parse tree is inefficient

Example:

Parse Tree for z = x + 2*y;



Finding a More Efficient Tree

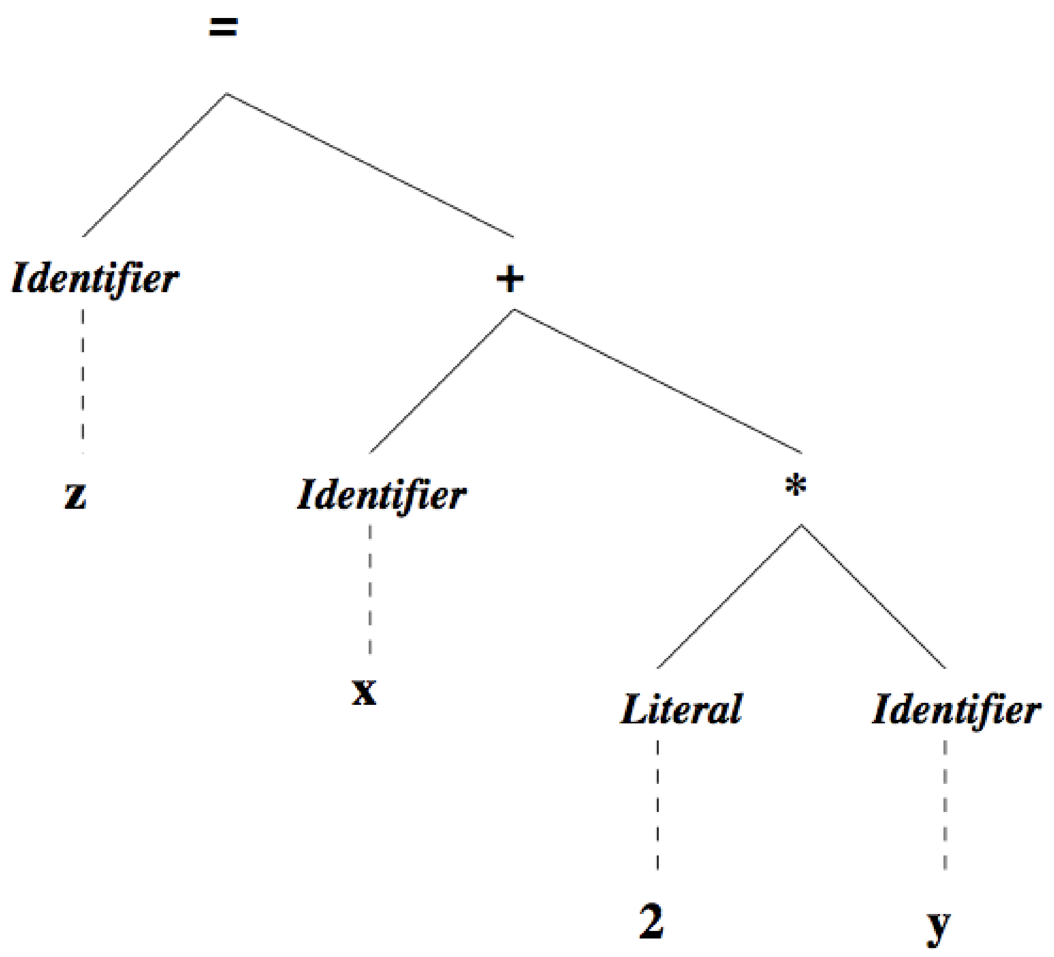
The *shape* of the parse tree reveals the meaning of the program.

So we want a tree that removes its inefficiency and keeps its shape.

- *Remove separator/punctuation terminal symbols*
- *Remove all trivial root nonterminals*
- *Replace remaining nonterminals with leaf terminals*

Example:

Abstract Syntax Tree for z = x + 2*y;



Abstract Syntax

Removes “syntactic sugar” and keeps essential elements of a language. E.g., consider the following two equivalent loops:

<u>Pascal</u>

```
while i < n do begin
    i := i + 1;
end;
```

<u>C/C++</u>

```
while (i < n) {
    i = i + 1;
}
```

The only essential information in each of these is 1) that it is a *loop*, 2) that its terminating condition is `i < n`, and 3) that its body increments the current value of `i`.

Abstract Syntax

```
#Concrete syntax        #Abstract syntax
(3+4)                   Add(Number(3) + Number(4))
```

```
#Concrete syntax        #Abstract syntax
x = y +1                Assign(
                               Var(x),
                               Add(Var(y), Number(2))
                        )
```

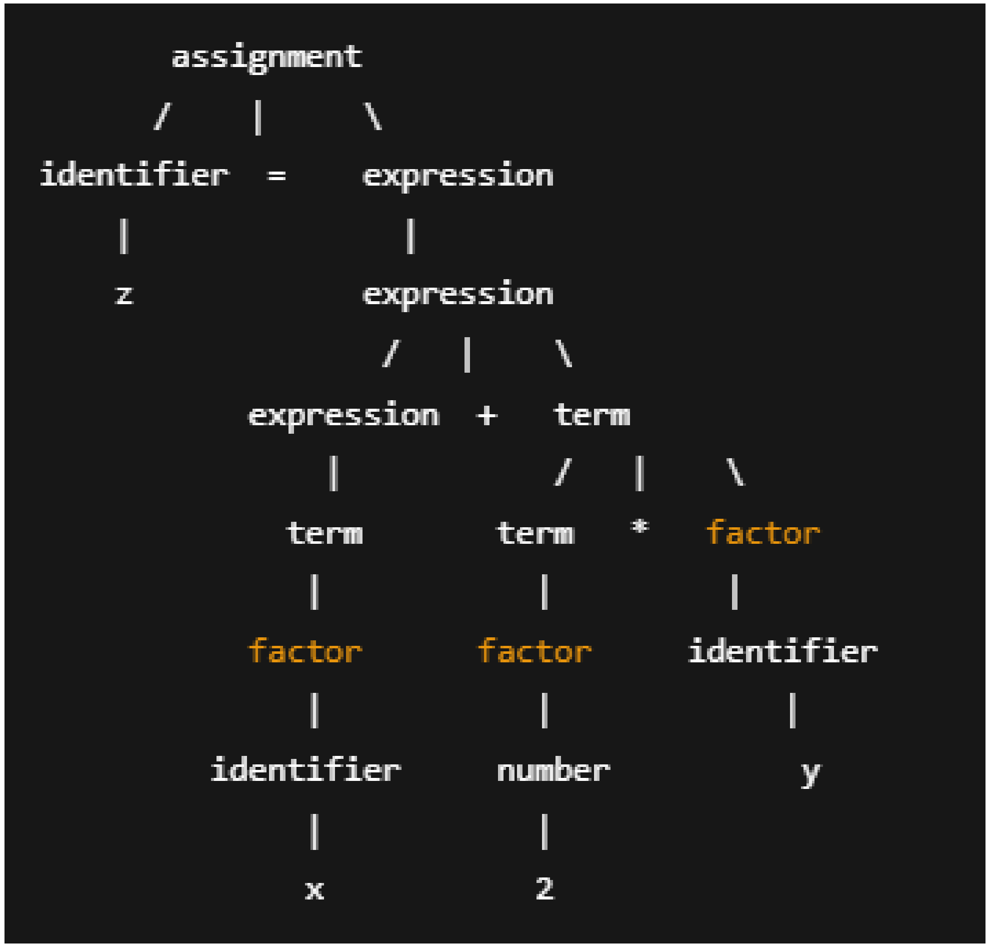
```
#Concrete syntax        #Abstract syntax
(3+4)                   Add(Number(3) + Number(4))
3+4
((3) + (4))
```

Abstract Syntax Tree (AST)

- **Concrete syntax** defines *how* expressions are written according to the grammar.
- **Abstract syntax** defines the *essential elements* of expressions, independent of grammar details.
- An **Abstract Syntax Tree (AST)** removes unnecessary syntactic nodes and keeps only semantic structure.

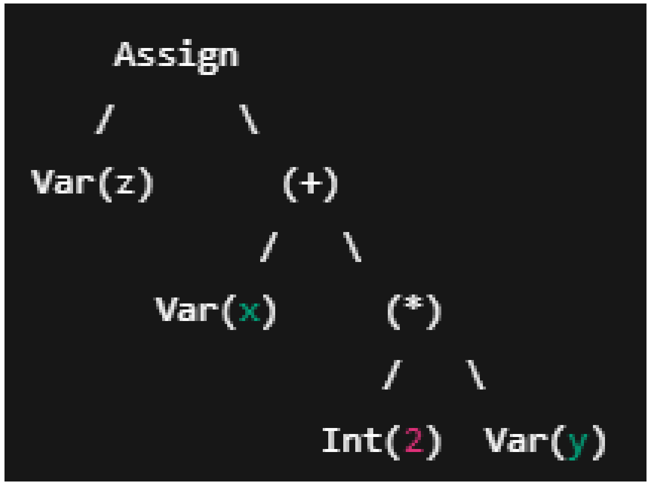
$$z = x + 2 * y;$$

Concrete Syntax Tree



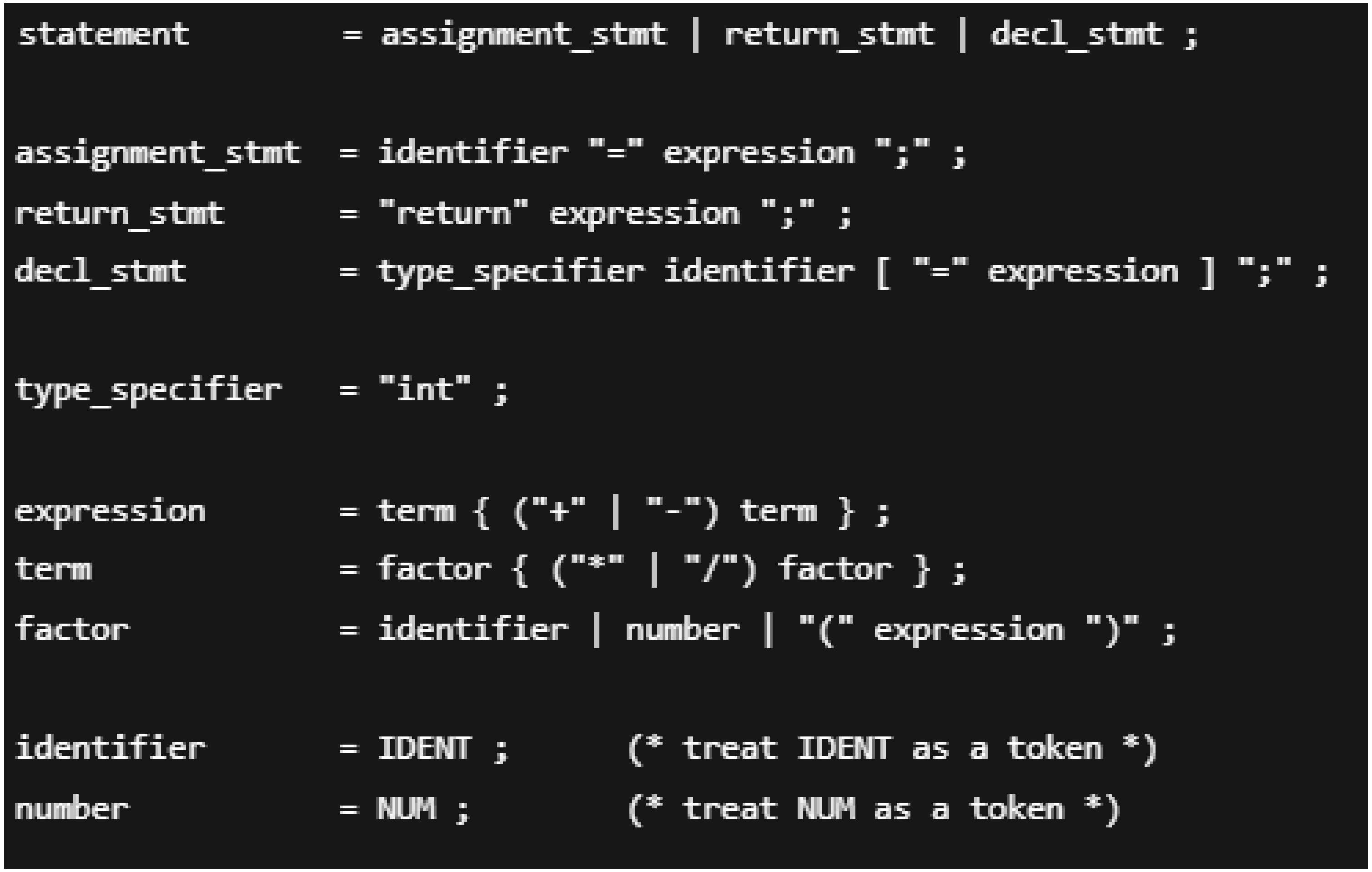
```
       assignment
      /    |     \
identifier  =    expression
    |              |
    z            expression
                  /   |    \
           expression  +   term
               |           /   |    \
             term       term   *   factor
              |           |         |
           factor      factor     identifier
              |           |            |
         identifier     number          y
              |           |
              x           2
```

AST



```
    Assign
   /      \
Var(z)      (+)
            /   \
      Var(x)     (*)
                  /   \
              Int(2)  Var(y)
```

Exercise



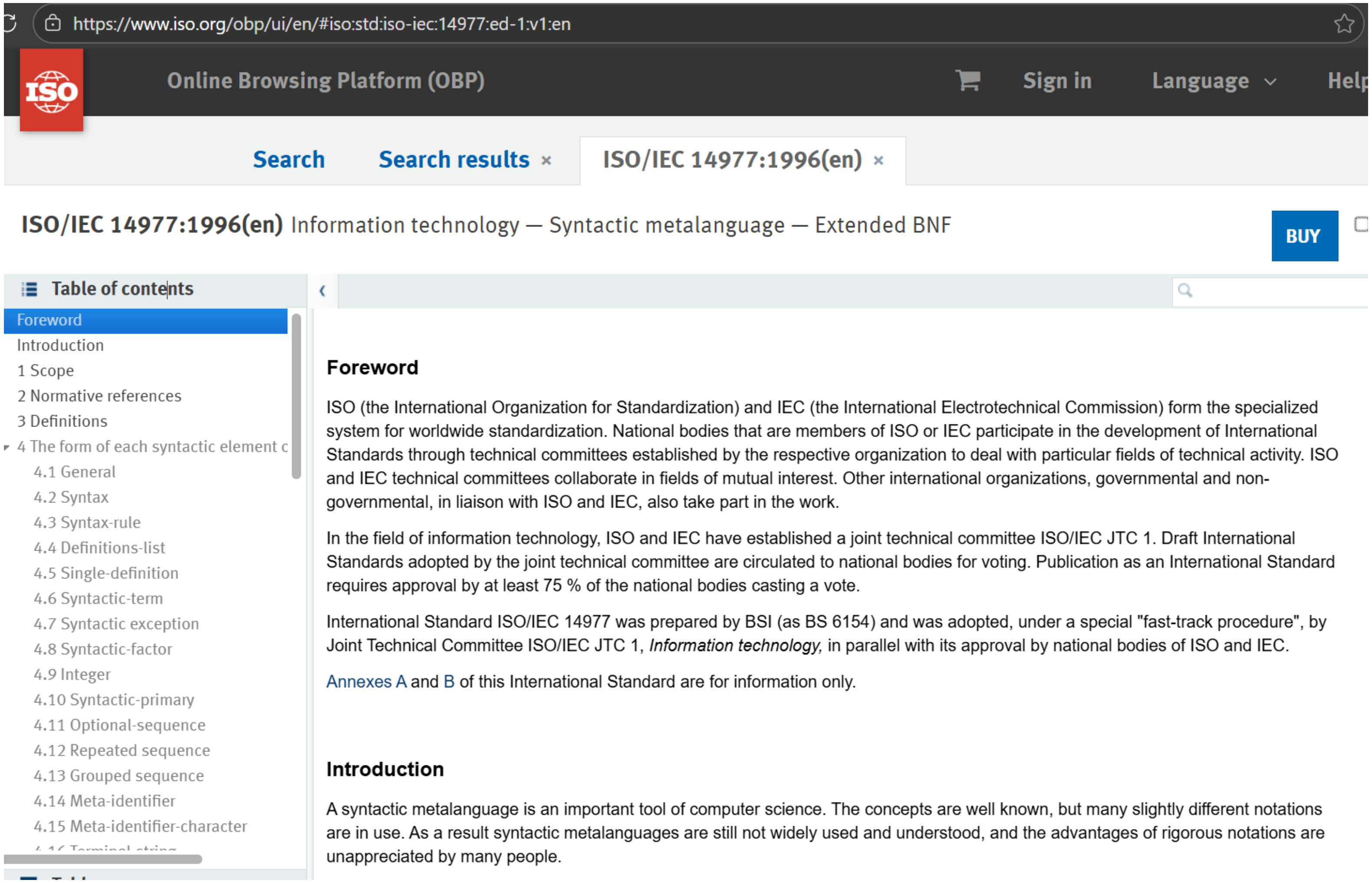
```
statement        = assignment_stmt | return_stmt | decl_stmt ;

assignment_stmt  = identifier "=" expression ";" ;
return_stmt      = "return" expression ";" ;
decl_stmt        = type_specifier identifier [ "=" expression ] ";" ;

type_specifier   = "int" ;

expression       = term { ("+" | "-") term } ;
term             = factor { ("*" | "/") factor } ;
factor           = identifier | number | "(" expression ")" ;

identifier       = IDENT ;      (* treat IDENT as a token *)
number           = NUM ;        (* treat NUM as a token *)
```

Show leftmost derivation of each statement using the EBNF

```
x = 3;

return x + 2 * y;

int z = (x + 2) * y;
```

Appendix

More details of EBNF



https://www.iso.org/obp/ui/en/#iso:std:iso-iec:14977:ed-1:v1:en

ISO/IEC 14977:1996(en) Information technology — Syntactic metalanguage — Extended BNF

Table of contents

Foreword
Introduction
1 Scope
2 Normative references
3 Definitions
4 The form of each syntactic element c
4.1 General
4.2 Syntax
4.3 Syntax-rule
4.4 Definitions-list
4.5 Single-definition
4.6 Syntactic-term
4.7 Syntactic exception
4.8 Syntactic-factor
4.9 Integer
4.10 Syntactic-primary
4.11 Optional-sequence
4.12 Repeated sequence
4.13 Grouped sequence
4.14 Meta-identifier
4.15 Meta-identifier-character

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

In the field of information technology, ISO and IEC have established a joint technical committee ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

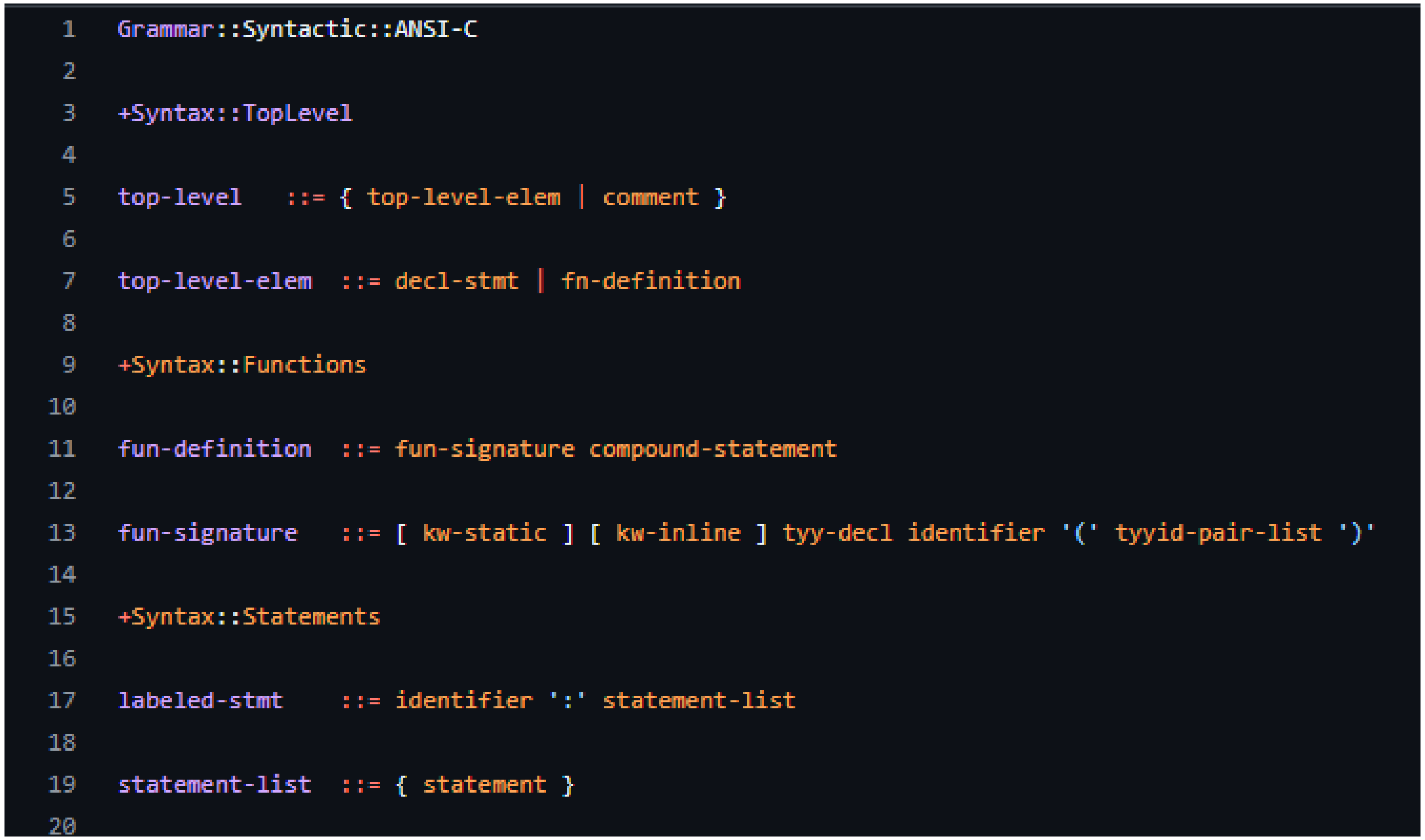
International Standard ISO/IEC 14977 was prepared by BSI (as BS 6154) and was adopted, under a special "fast-track procedure", by Joint Technical Committee ISO/IEC JTC 1, *Information technology,* in parallel with its approval by national bodies of ISO and IEC.

Annexes A and B of this International Standard are for information only.

Introduction

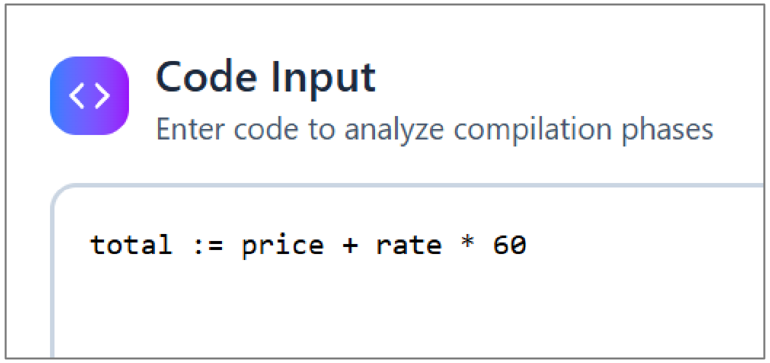
A syntactic metalanguage is an important tool of computer science. The concepts are well known, but many slightly different notations are in use. As a result syntactic metalanguages are still not widely used and understood, and the advantages of rigorous notations are unappreciated by many people.

EBNF for C



```
Grammar::Syntactic::ANSI-C

+Syntax::TopLevel

top-level    ::= { top-level-elem | comment }

top-level-elem  ::= decl-stmt | fn-definition

+Syntax::Functions

fun-definition  ::= fun-signature compound-statement

fun-signature   ::= [ kw-static ] [ kw-inline ] tyy-decl identifier '(' tyyid-pair-list ')'

+Syntax::Statements

labeled-stmt    ::= identifier ':' statement-list

statement-list  ::= { statement }

```

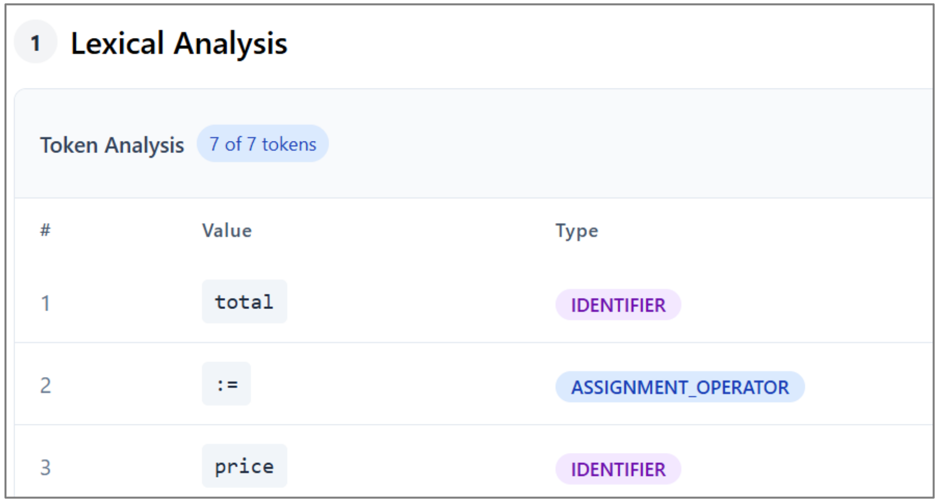
https://gist.github.com/Chubek/52884d1fa766fa16ae8d8f226ba105ad



Code Input

Enter code to analyze compilation phases

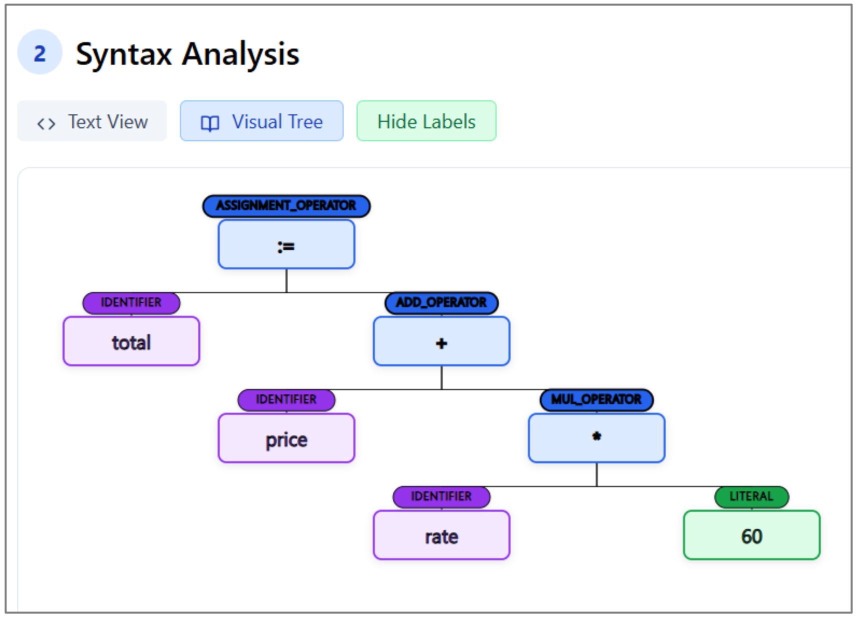
```
total := price + rate * 60
```

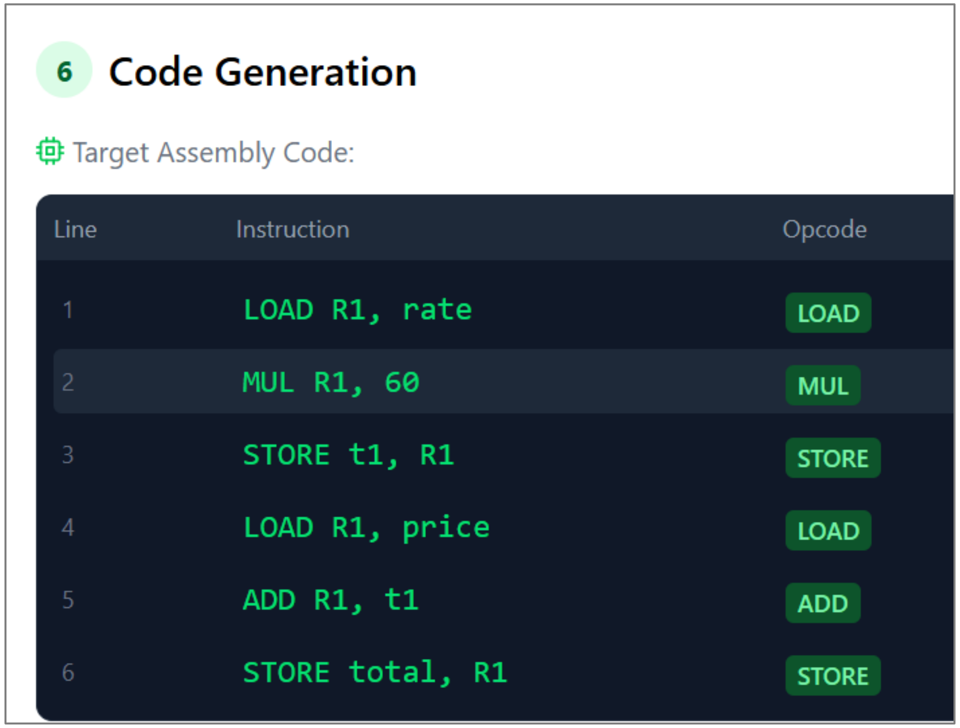


1 Lexical Analysis

Token Analysis 7 of 7 tokens

#	Value	Type
1	`total`	IDENTIFIER
2	`:=`	ASSIGNMENT_OPERATOR
3	`price`	IDENTIFIER



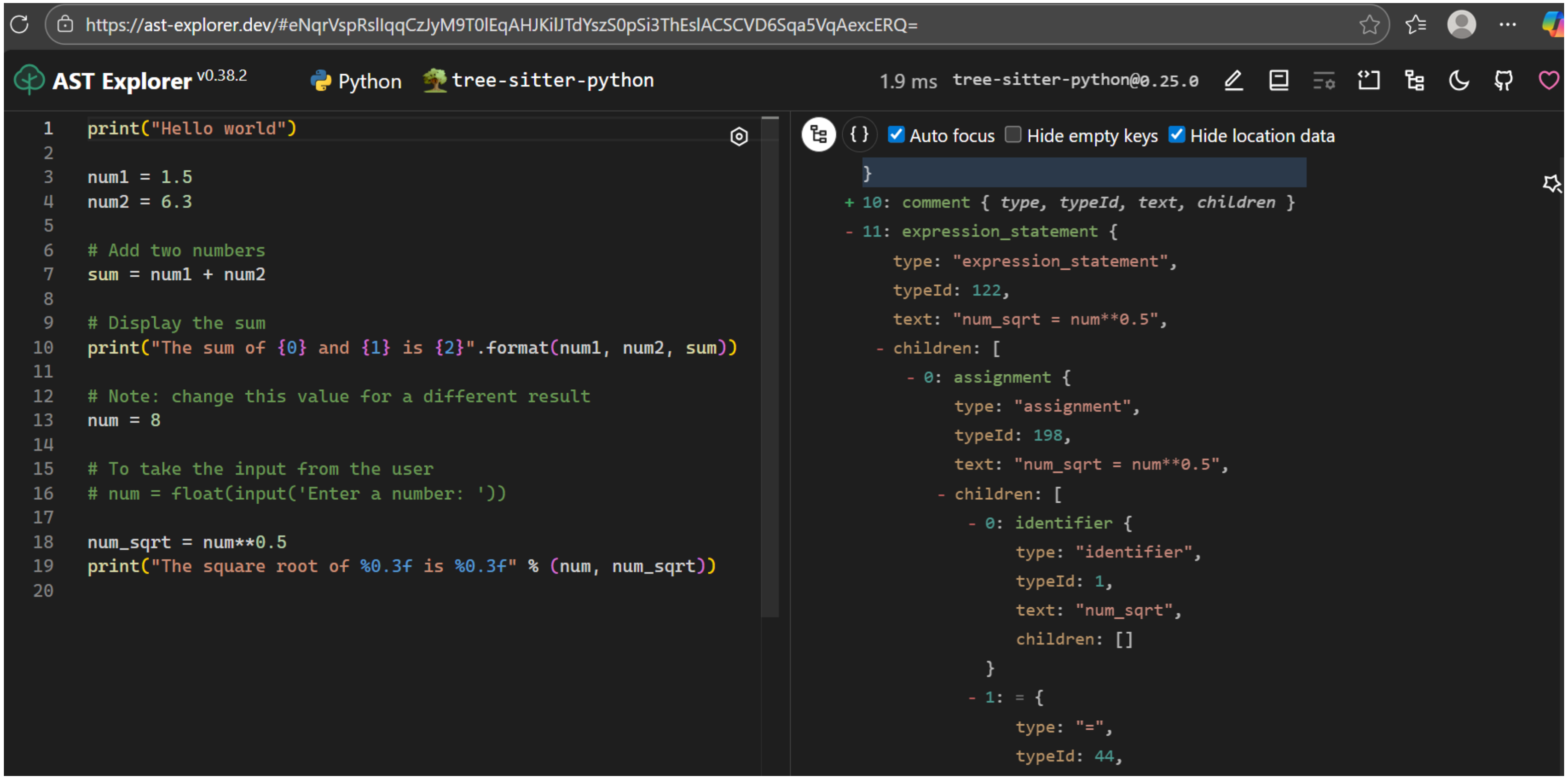


6 Code Generation

Target Assembly Code:

Line	Instruction	Opcode
1	`LOAD R1, rate`	LOAD
2	`MUL R1, 60`	MUL
3	`STORE t1, R1`	STORE
4	`LOAD R1, price`	LOAD
5	`ADD R1, t1`	ADD
6	`STORE total, R1`	STORE

https://compiler-visualizer-seven.vercel.app/


https://ast-explorer.dev/#eNqrVspRslIqqCzJyM9T0lEqAHJKiIJTdYszS0pSi3ThEslACSCVD6Sqa5VqAexcERQ=
AST Explorer v0.38.2
Python
tree-sitter-python
1.9 ms
tree-sitter-python@0.25.0
Auto focus
Hide empty keys
Hide location data

```python
print("Hello world")

num1 = 1.5
num2 = 6.3

# Add two numbers
sum = num1 + num2

# Display the sum
print("The sum of {0} and {1} is {2}".format(num1, num2, sum))

# Note: change this value for a different result
num = 8

# To take the input from the user
# num = float(input('Enter a number: '))

num_sqrt = num**0.5
print("The square root of %0.3f is %0.3f" % (num, num_sqrt))
```

```
}
+ 10: comment { type, typeId, text, children }
- 11: expression_statement {
    type: "expression_statement",
    typeId: 122,
    text: "num_sqrt = num**0.5",
  - children: [
    - 0: assignment {
        type: "assignment",
        typeId: 198,
        text: "num_sqrt = num**0.5",
      - children: [
        - 0: identifier {
            type: "identifier",
            typeId: 1,
            text: "num_sqrt",
            children: []
          }
        - 1: = {
            type: "=",
            typeId: 44,
```

https://ast-explorer.dev/