Programming Languages

2nd edition

Tucker and Noonan

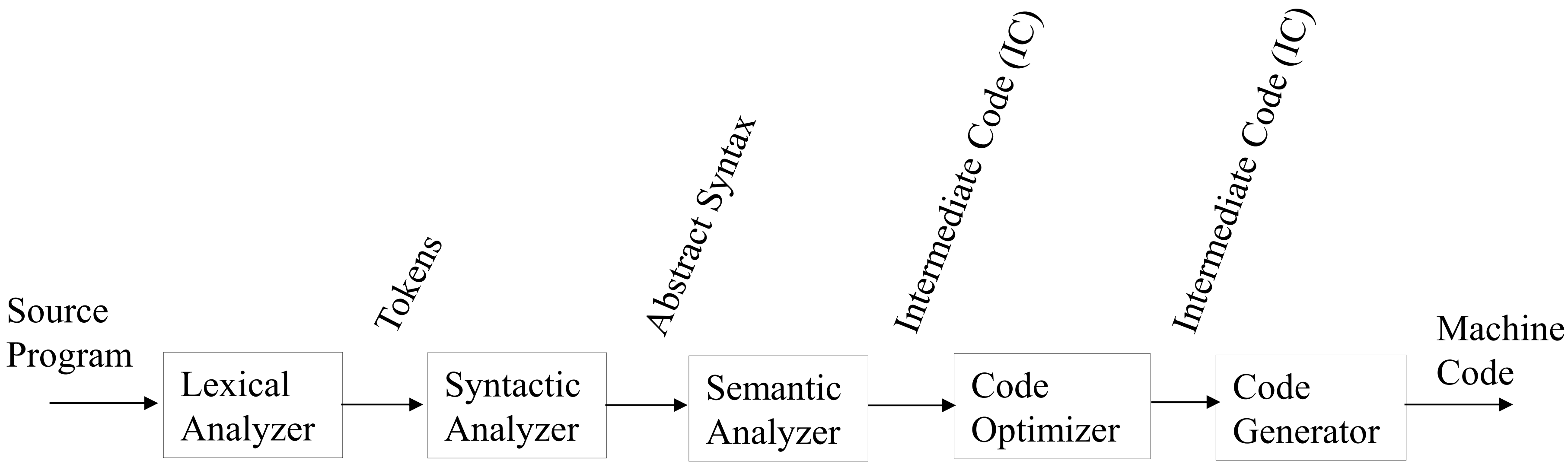
Chapter 2
Syntax

A language that is simple to parse for the compiler is also simple to parse for the human programmer.

N. Wirth

Contents

2.4 Compilers and Interpreters



Lexer

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

Parser

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

Semantic Analysis

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

Code Optimization

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

Code Generation

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

Interpreter

Replaces last 2 phases of a compiler

Input:

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

Mixed interpreters

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

Pure interpreters:

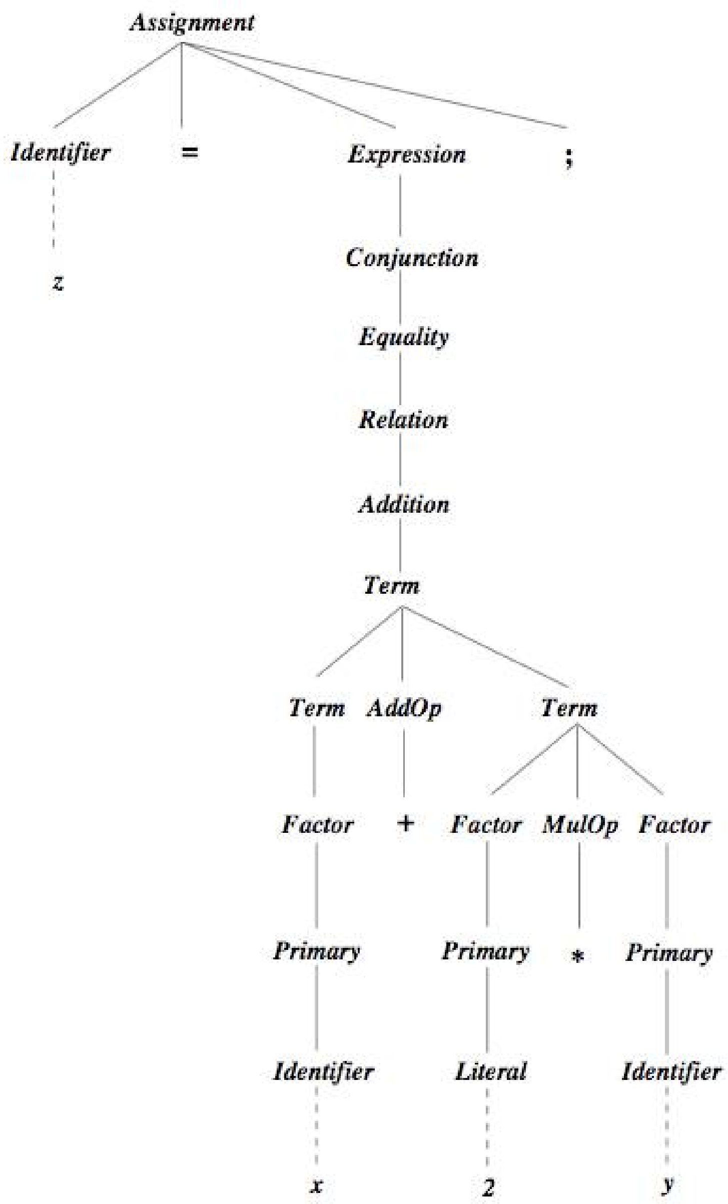
- *most Basics, shell commands*

2.5 Linking Syntax and Semantics

Output: parse tree is inefficient

Example: Fig. 2.9

Parse Tree for
z = x + 2*y;
Fig. 2.9



Finding a More Efficient Tree

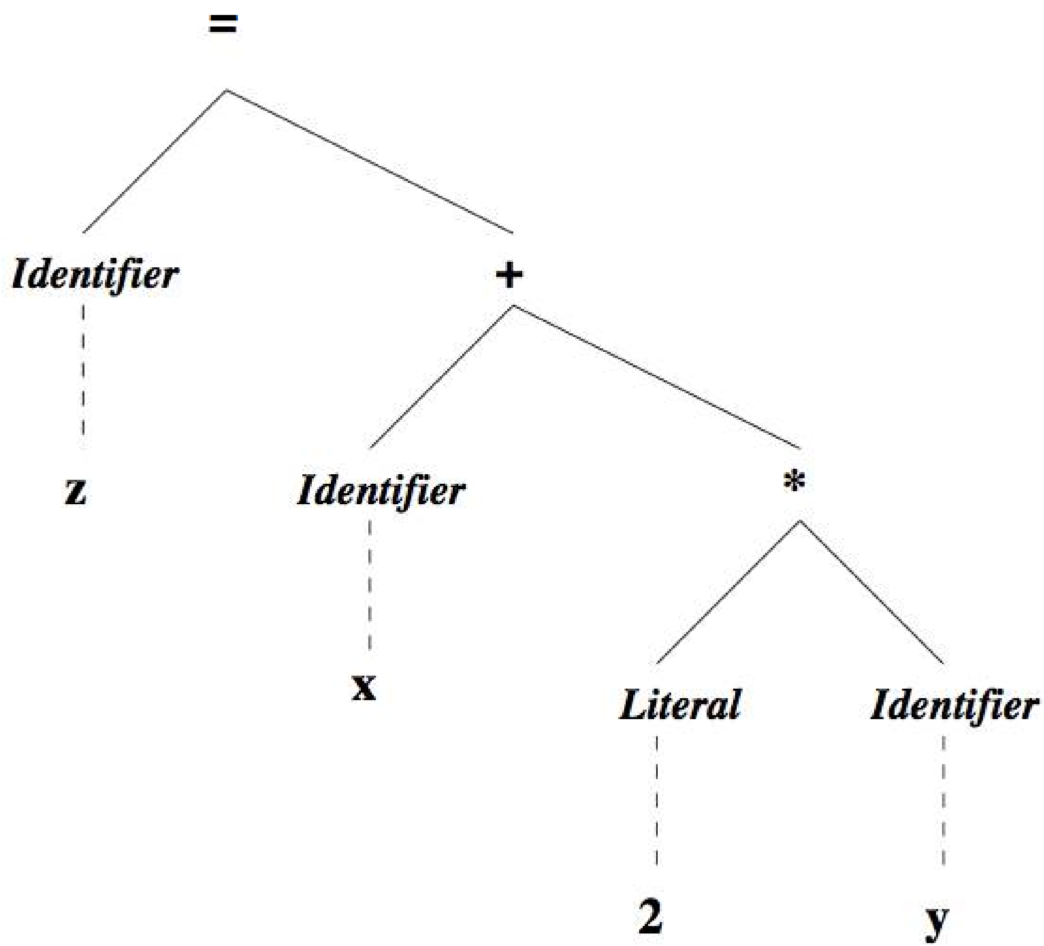
The *shape* of the parse tree reveals the meaning of the program.

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: Fig. 2.10

Abstract Syntax Tree for
z = x + 2*y;
Fig. 2.10



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 of *Clite* Assignments

Assignment = Variable target*; Expression* source

Expression = VariableRef | Value | Binary | Unary

VariableRef = Variable | ArrayRef

Variable = String id

ArrayRef = String id*; Expression* index

Value = IntValue | BoolValue | FloatValue | CharValue

Binary = Operator op*; Expression* term1, term2

Unary = UnaryOp op*; Expression* term

Operator = ArithmeticOp | RelationalOp | BooleanOp

IntValue = Integer intValue

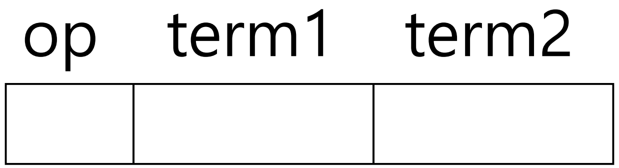
…

Abstract Syntax as Java Classes

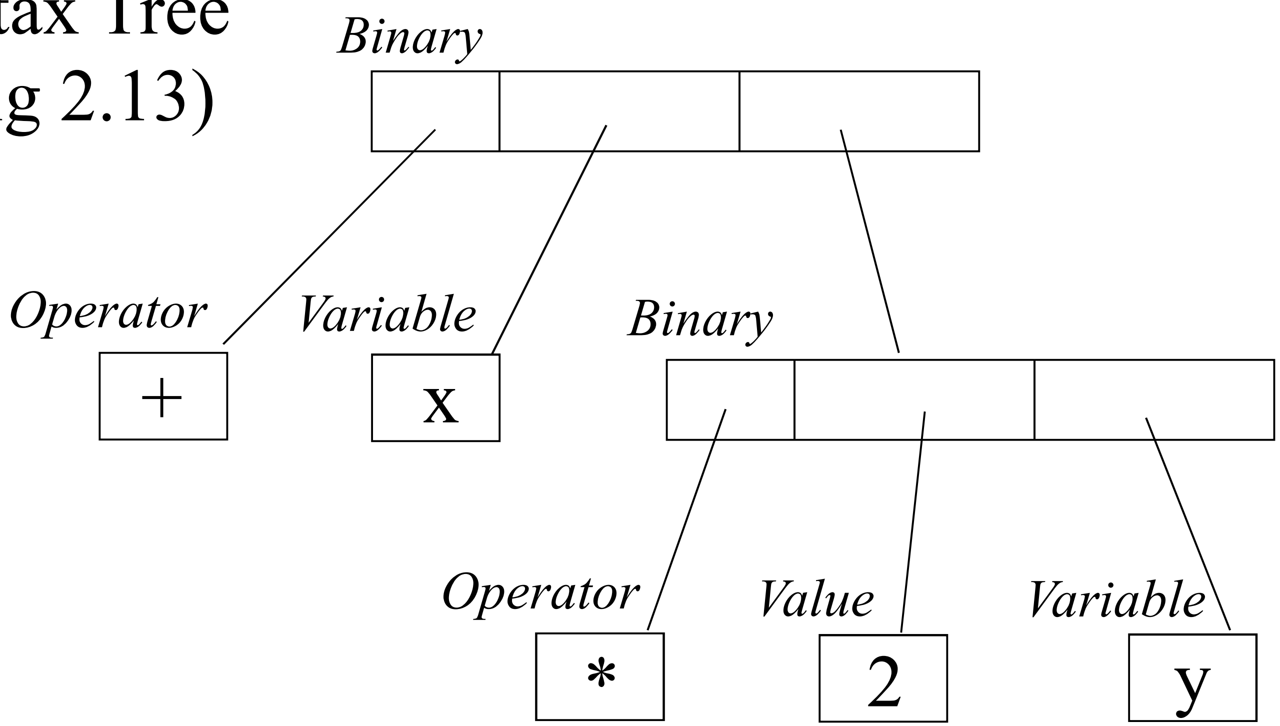
```
abstract class Expression { }
abstract class VariableRef extends Expression { }
class Variable extends VariableRef { String id; }
class Value extends Expression { … }
class Binary extends Expression {
Operator op;
Expression term1, term2;
}
class Unary extends Expression {
UnaryOp op;
Expression term;
}
```

Example Abstract Syntax Tree

Binary node



Abstract Syntax Tree
for x+2*y (Fig 2.13)



Remaining Abstract Syntax of *Clite* (*Declarations* and *Statements*)

Fig 2.14

$$
\begin{aligned}
Program &= Declarations\ \texttt{decpart};\ \ Statements\ \texttt{body}; \\
Declarations &= Declaration^* \\
Declaration &= VariableDecl \mid ArrayDecl \\
VariableDecl &= Variable\ \texttt{v};\ Type\ \texttt{t} \\
ArrayDecl &= Variable\ \texttt{v};\ Type\ \texttt{t};\ Integer\ \texttt{size} \\
Type &= \ \texttt{int} \mid \texttt{bool} \mid \texttt{float} \mid \texttt{char} \\
Statements &= Statement^* \\
Statement &= Skip \mid Block \mid Assignment \mid Conditional \mid Loop \\
Skip &= \\
Block &= Statements \\
Conditional &= Expression\ \texttt{test};\ Statement\ \texttt{thenbranch, elsebranch} \\
Loop &= Expression\ \texttt{test};\ Statement\ \texttt{body}
\end{aligned}
$$