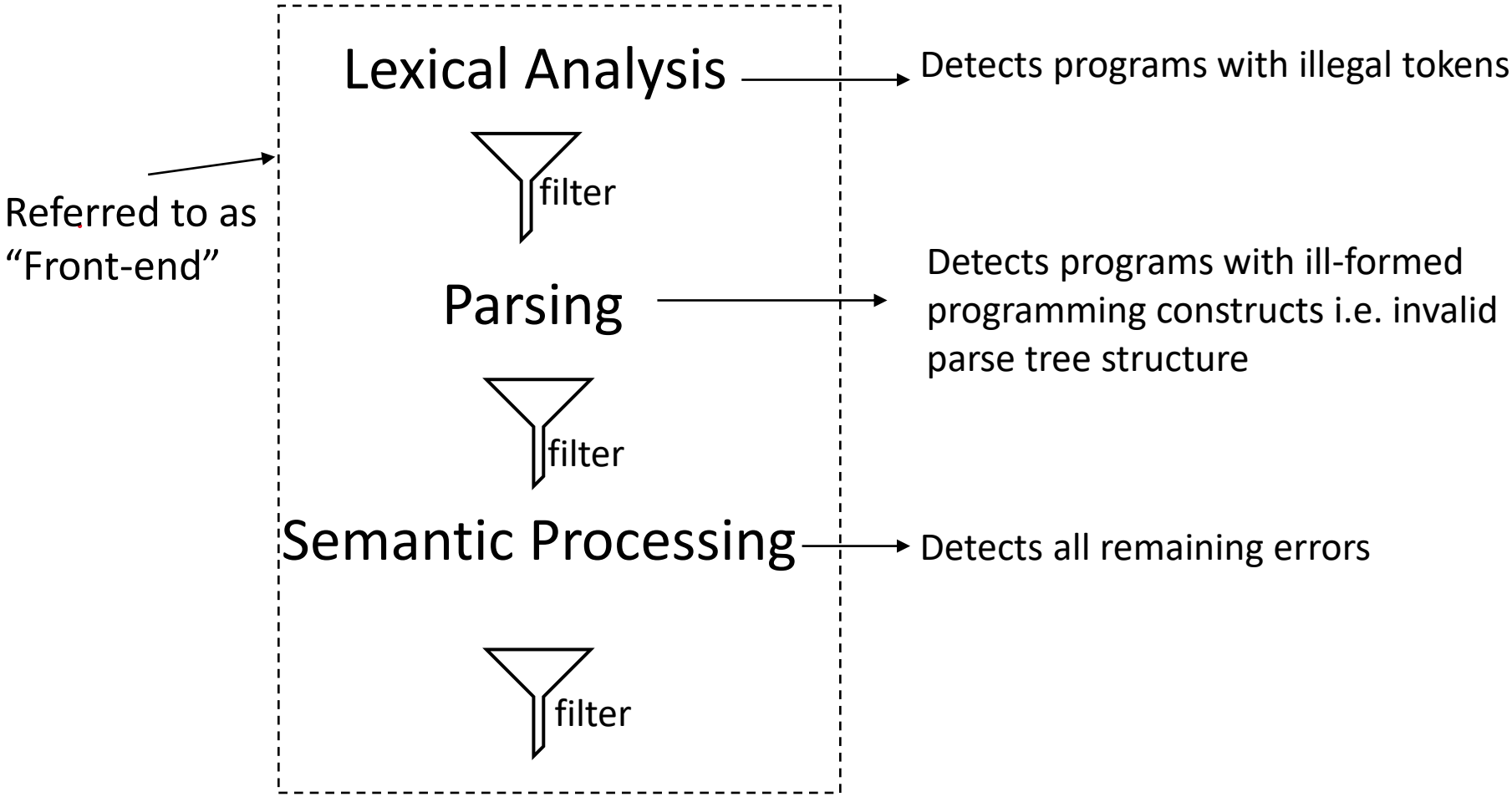
CS323: Compilers

Spring 2023

Week 5: Parsers (discussion and conclusion), Semantic Routines

Semantic Processing



Semantic Processing

- Syntax-directed / syntax-driven
  - Routines (called as semantic routines) interpret the meaning of programming constructs based on the syntactic structure
  - Routines play a dual role
    - Analysis – Semantic analysis
      - undefined vars, undefined types, uninitialized variables, type errors that can be caught at compile time, unreachable code, etc.
    - Synthesis – Generation of intermediate code
      - 3 address code
  - Routines create semantic records to aid the analysis and synthesis

Semantic Processing

- Syntax-directed translation: notation for *attaching* program fragments to grammar productions.
  - Program fragments are executed when productions are matched
  - The combined execution of all program fragments produces the translation of the program

    e.g. `E->E+T  { print('+') }`

`Output:` program fragments may create AST and 3 Address Codes

- Attributes: any 'quality' associated with a terminal and non-terminal e.g. type, number of lines of a code, first line of the code block etc.

Why Semantic Analysis?

- Context-free grammars cannot specify all requirements of a language
  - Identifiers declared before their use (scope)
  - Types in an expression must be consistent

    STRING str:= “Hello”;

    str := str + 2;
  - Number of formal and actual parameters of a function must match
  - Reserved keywords cannot be used as identifiers
  - A Class is declared only once in a OO language program, a method of a class can be overridden.
  - …

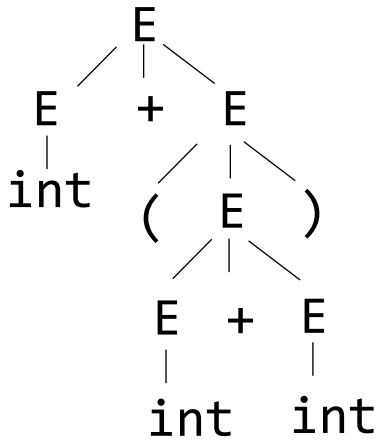
Abstract Syntax Tree

- Abstract Syntax Tree (AST) or Syntax Tree can be the input for semantic analysis.
  - What is Concrete Syntax Tree? – the parse tree
- ASTs are like parse trees but ignore certain details:

E.g. Consider the grammar:

```
E -> E + E
   | ( E )
   | int
```

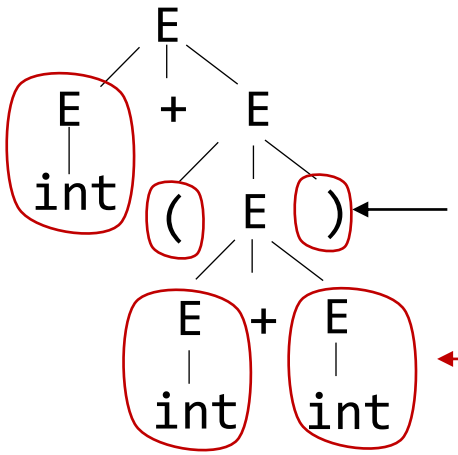
The parse tree for 1+(2+3)



AST - Example

- Not all details (nodes) of the parse tee are helpful for semantic analysis

The *parse tree* for `1+(2+3)` :

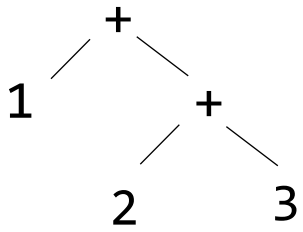


Expresses associativity. Lower subtree in the hierarchy can express.

Single child. Can compress.

We need to compute the result of the expression. So, a simpler structure is sufficient:

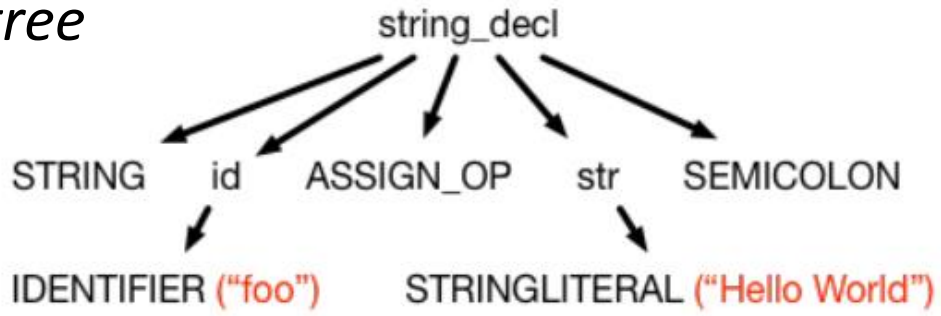
AST for `1+(2+3)` :



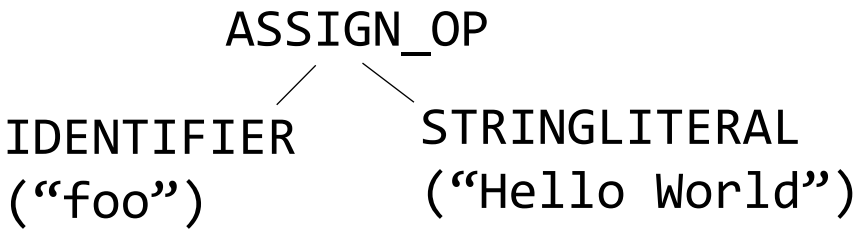
AST - Example

```
STRING foo := “Hello World” ;
```

parse tree ⇨



≡ *AST* ⇩



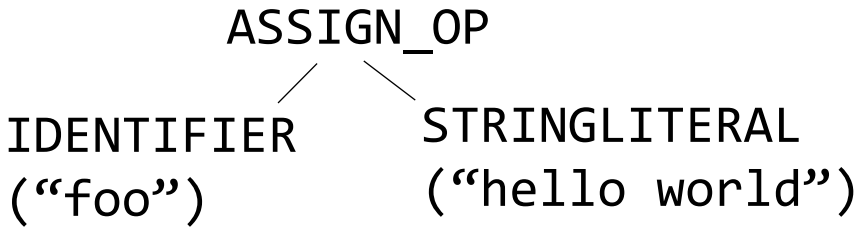
Semantic Analysis – Example

- Context-free grammars cannot specify all requirements of a language
  - Identifiers declared before their use (scope)
  - Types in an expression must be consistent

    Type checks
    STRING str:= “Hello”;
    str := str + 2;
  - Number of formal and actual parameters of a function must match
  - Reserved keywords cannot be used as identifiers
  - A Class is declared only once in a OO language, a method can be overridden.
  - …

Scope

- **Goal:** matching identifier declarations with uses
- Most languages require this!
- Scope confines the activity of an identifier



What if `foo` is declared as a STRING in an enclosing scope but is an INT in the current scope?

in different parts of the program:

- Same identifier may refer to different things
- Same identifier may not be accessible

Static Scope

- Most languages are statically scoped
  - Scope depends on only the program text (not runtime behavior)
  - A variable refers to the closest defined instance

```
INT w, x;
{
    FLOAT x, z;
    f(x, w, z);
}
g(x)
```

x is a FLOAT here

x is an INT here

Dynamic Scope

- In dynamically scoped languages
  - Scope depends on the execution context
  - A variable refers to the closest enclosing binding in the execution of the program

```
f(){
    a=4; g();
}
g() { print(a); }
```

value of a is 4 here

Exercise: Static vs. Dynamic Scope

```
#define a (x+1) //macro definition

int x = 2; //global var definition

//function b definition
void b() {
    int x = 1;
    printf("%d\n",a);
}

//function c definition
void c() {
    printf("%d\n",a);
}

//the main function
int main() { b(); c(); }
```

Is x statically scoped or dynamically scoped?