Example

Let's look at the CUP specification for CSX-lite. Recall its CFG is

program $\rightarrow$ **{ stmts }**
stmts $\rightarrow$ **stmt stmts**
| λ
stmt $\rightarrow$ **id = expr ;**
| **if (expr) stmt**
expr $\rightarrow$ **expr + id**
| **expr - id**
| **id**

The corresponding CUP specification is:

```
/***
This Is A Java CUP Specification For
CSX-lite, a Small Subset of The CSX
Language,  Used In Cs536
 ***/


/* Preliminaries to set up and use the
scanner.  */


import java_cup.runtime.*;
parser code {:
 public void syntax_error
   (Symbol cur_token){
    report_error(
     "CSX syntax error at line "+
     String.valueOf(((CSXToken)
        cur_token.value).linenum),
     null);}
:};

init with {:              :};
scan with {:
    return Scanner.next_token();
:};
```

```
/* Terminals (tokens returned by the
scanner). */
terminal CSXIdentifierToken IDENTIFIER;
terminal CSXToken SEMI, LPAREN, RPAREN,
ASG, LBRACE, RBRACE;
terminal CSXToken PLUS, MINUS, rw_IF;

/* Non terminals */
non terminal csxLiteNode prog;
non terminal stmtsNode   stmts;
non terminal stmtNode    stmt;
non terminal exprNode    exp;
non terminal nameNode    ident;


start with prog;

prog::= LBRACE:l stmts:s RBRACE
 {: RESULT=
      new csxLiteNode(s,
           l.linenum,l.colnum); :}
;

stmts::= stmt:s1  stmts:s2
 {: RESULT=
      new stmtsNode(s1,s2,
        s1.linenum,s1.colnum);
 :}
```

```
|
 {: RESULT= stmtsNode.NULL; :}
;
stmt::= ident:id ASG exp:e SEMI
 {: RESULT=
       new asgNode(id,e,
           id.linenum,id.colnum);
 :}

| rw_IF:i LPAREN exp:e RPAREN  stmt:s
 {: RESULT=new ifThenNode(e,s,
            stmtNode.NULL,
            i.linenum,i.colnum); :}
;
exp::=
 exp:leftval PLUS:op ident:rightval
 {: RESULT=new binaryOpNode(leftval,
      sym.PLUS, rightval,
      op.linenum,op.colnum); :}

| exp:leftval MINUS:op ident:rightval
 {: RESULT=new binaryOpNode(leftval,
           sym.MINUS,rightval,
           op.linenum,op.colnum); :}
| ident:i
 {: RESULT = i; :}
;
```

```
ident::= IDENTIFIER:i
 {: RESULT = new nameNode(
   new identNode(i.identifierText,
                 i.linenum,i.colnum),
   exprNode.NULL,
   i.linenum,i.colnum); :}
;
```

Let's parse

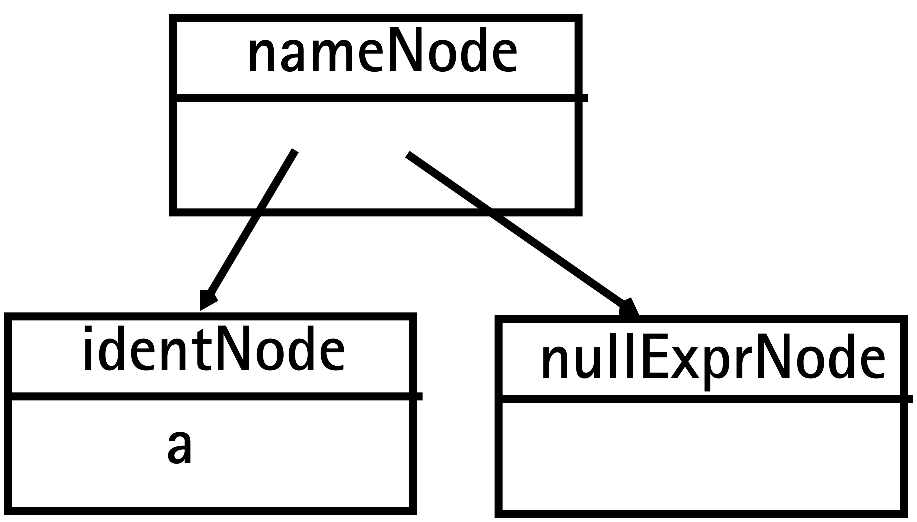
{ a = b ; }

First, **a** is parsed using

```
ident::= IDENTIFIER:i
 {: RESULT = new nameNode(
   new identNode(i.identifierText,
                 i.linenum,i.colnum),
   exprNode.NULL,

   i.linenum,i.colnum); :}
```

We build

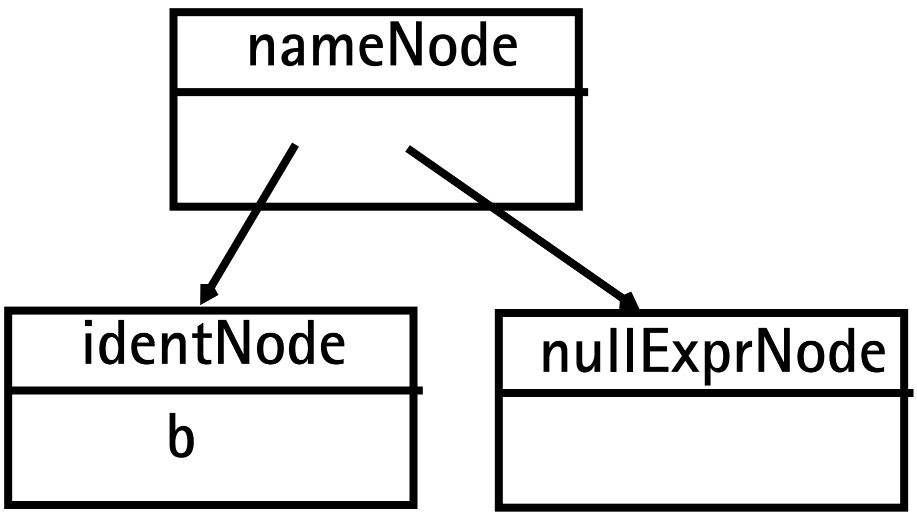


Next, **b** is parsed using

```
ident::= IDENTIFIER:i
 {: RESULT = new nameNode(
    new identNode(i.identifierText,
                  i.linenum,i.colnum),
    exprNode.NULL,

    i.linenum,i.colnum); :}
```

We build



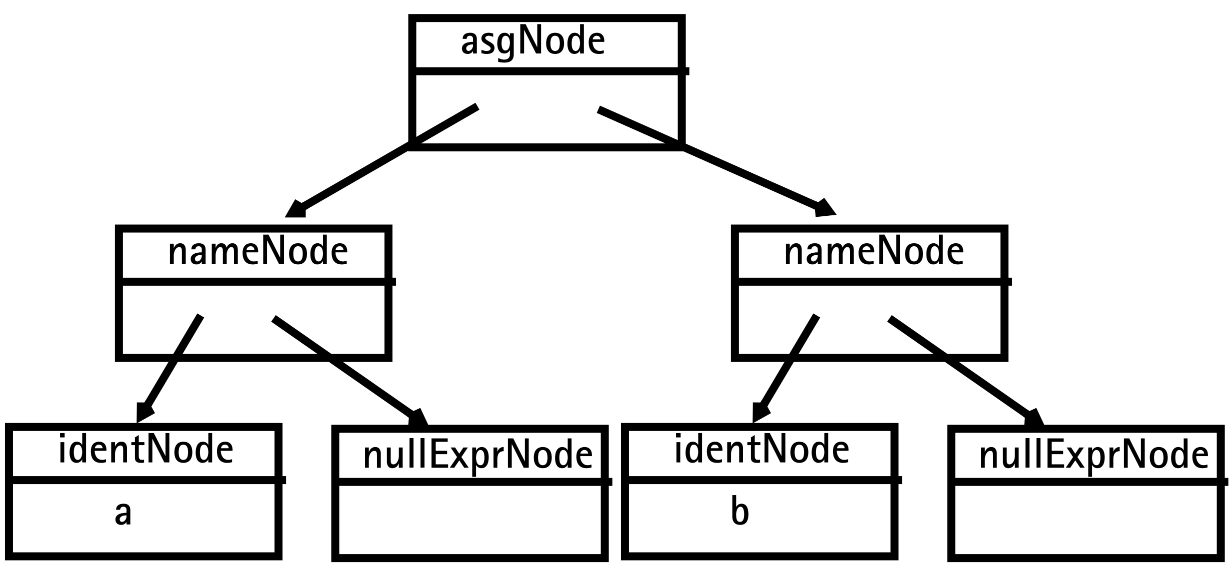
Then `b`'s subtree is recognized as an **exp**:

```
| ident:i
 {: RESULT = i; :}
```

Now the assignment statement is recognized:

```
stmt::= ident:id ASG exp:e SEMI
 {: RESULT=
        new asgNode(id,e,
            id.linenum,id.colnum);
 :}
```

We build

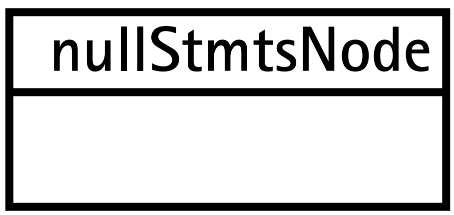


The **stmts** $\rightarrow$ λ production is matched (indicating that there are no more statements in the program).

CUP matches

```
stmts::=
 {: RESULT= stmtsNode.NULL; :}
```

and we build



Next,

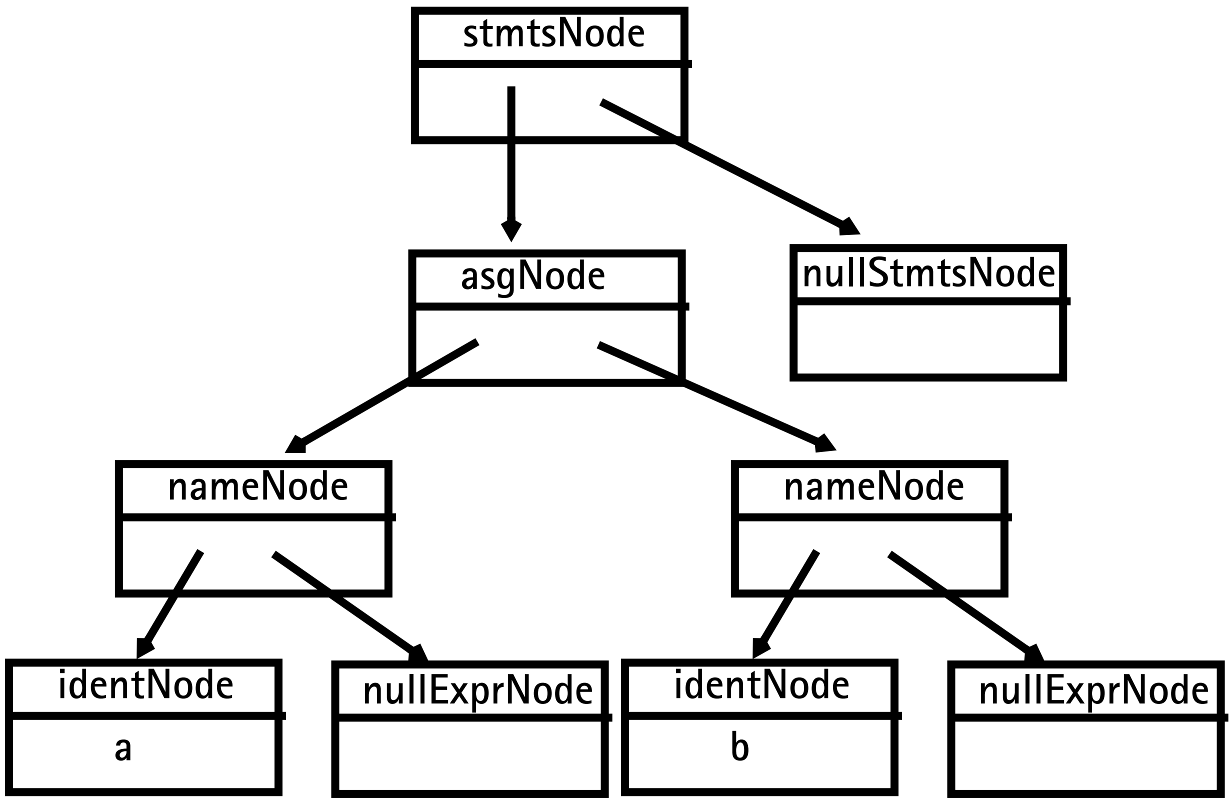
stmts $\rightarrow$ **stmt** **stmts**

is matched using

```
stmts::= stmt:s1  stmts:s2
 {: RESULT=
     new stmtsNode(s1,s2,
       s1.linenum,s1.colnum);

 :}
```

This builds



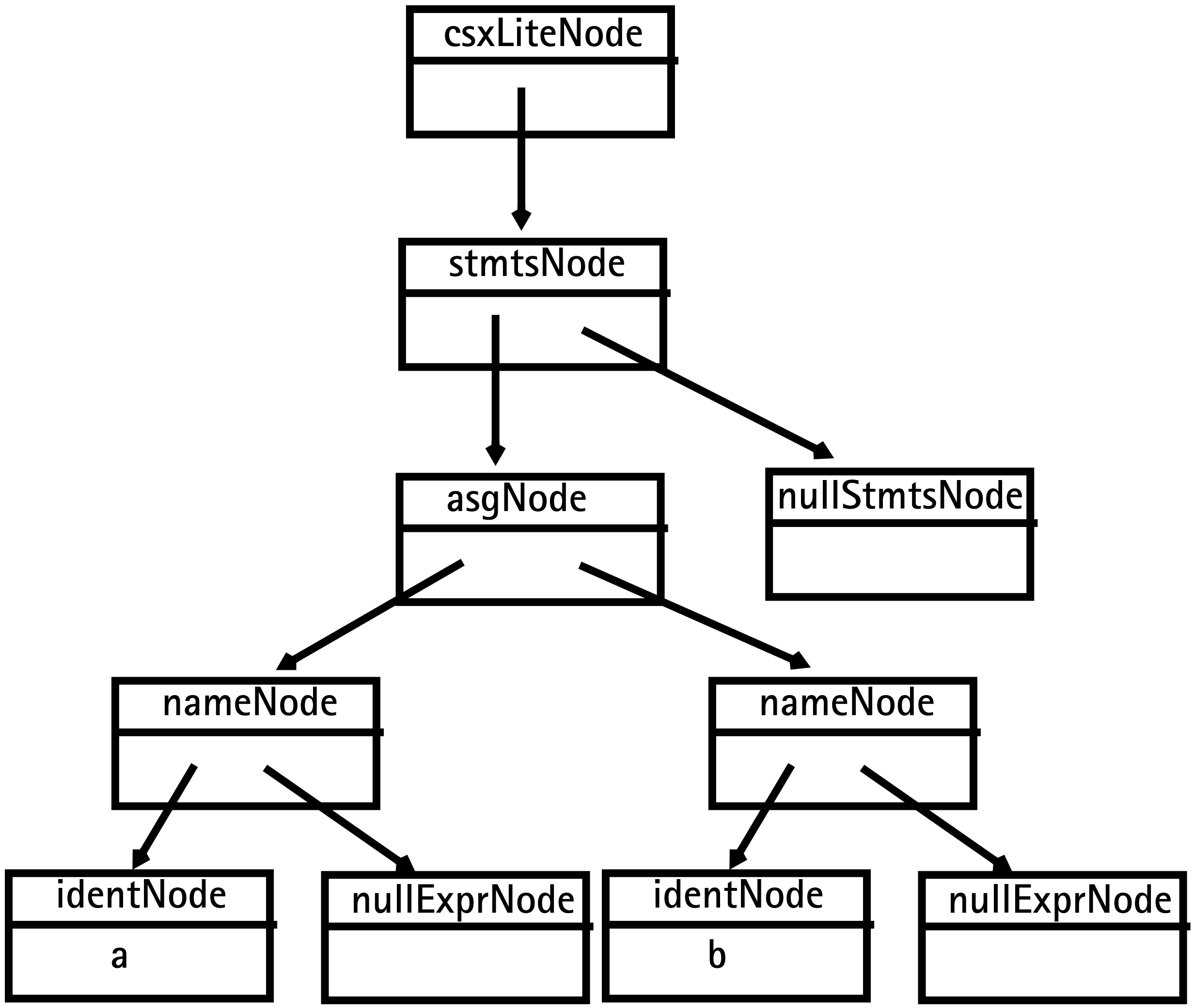
As the last step of the parse, the parser matches

program $\rightarrow$ **{ stmts }**

using the CUP rule

```
prog::= LBRACE:l stmts:s RBRACE
 {: RESULT=
      new csxLiteNode(s,
           l.linenum,l.colnum); :}
;
```

The final AST reurned by the parser is



Errors in Context-Free Grammars

Context-free grammars can contain errors, just as programs do. Some errors are easy to detect and fix; others are more subtle.

In context-free grammars we start with the start symbol, and apply productions until a terminal string is produced.

Some context-free grammars may contain *useless* non-terminals.

Non-terminals that are unreachable (from the start symbol) or that derive no terminal string are considered useless.

Useless non-terminals (and productions that involve them) can be safely removed from a grammar without changing the language defined by the grammar.

A grammar containing useless non-terminals is said to be *non-reduced*.

After useless non-terminals are removed, the grammar is *reduced*.

Consider

$$
\begin{aligned}
\mathbf{S} &\rightarrow \mathbf{A}\ \ \mathbf{B} \\
&\ \ |\ \ \mathbf{x} \\
\mathbf{B} &\rightarrow \mathbf{b} \\
\mathbf{A} &\rightarrow \mathbf{a}\ \ \mathbf{A} \\
\mathbf{C} &\rightarrow \mathbf{d}
\end{aligned}
$$

Which non-terminals are unreachable? Which derive no terminal string?

Finding Useless Non-terminals

To find non-terminals that can derive one or more terminal strings, we'll use a marking algorithm.

We iteratively mark terminals that can derive a string of terminals, until no more non-terminals can be marked. Unmarked non-terminals are useless.

(1) Mark all terminal symbols

(2) Repeat
  If all symbols on the righthand side of a production are marked
  Then mark the lefthand side
 Until no more non-terminals can be marked

We can use a similar marking algorithm to determine which non-terminals can be reached from the start symbol:

(1) Mark the Start Symbol

(2) Repeat
    If the lefthand side of a production is marked
    Then mark all non-terminals in the righthand side
  Until no more non-terminals can be marked

λ Derivations

When parsing, we'll sometimes need to know which non-terminals can derive λ. (λ is "invisible" and hence tricky to parse).

We can use the following marking algorithm to decide which non-terminals derive λ

(1) For each production $A \rightarrow \lambda$
        mark A

(2) Repeat
        If the entire righthand
            side of a production
            is marked
        Then mark the lefthand side
    Until no more non-terminals
        can be marked

As an example consider

$$S \rightarrow A\ B\ C$$

$$A \rightarrow a$$

$$B \rightarrow C\ D$$

$$D \rightarrow d$$

$$\quad \mid \lambda$$

$$C \rightarrow c$$

$$\quad \mid \lambda$$

Recall that compilers prefer an unambiguous grammar because a unique parse tree structure can be guaranteed for all inputs.

Hence a unique translation, guided by the parse tree structure, will be obtained.

We would like an algorithm that checks if a grammar is ambiguous.

Unfortunately, it is undecidable whether a given CFG is ambiguous, so such an algorithm is impossible to create.

Fortunately for certain grammar classes, including those for which we can generate parsers, we can prove included grammars are unambiguous.

Potentially, the most serious flaw that a grammar might have is that it generates the “wrong language."

This is a subtle point as a grammar serves as the *definition* of a language.

For established languages (like C or Java) there is usually a suite of programs created to test and validate new compilers. An incorrect grammar will almost certainly lead to incorrect compilations of test programs, which can be automatically recognized.

For new languages, initial implementors must thoroughly test the parser to verify that inputs are scanned and parsed as expected.

Parsers and Recognizers

Given a sequence of tokens, we can ask:

"Is this input syntactically valid?"

(Is it generable from the grammar?).

A program that answers this question is a *recognizer*.

Alternatively, we can ask:

"Is this input valid and, if it is, what is its structure (parse tree)?"

A program that answers this more general question is termed a *parser*.

We plan to use language structure to drive compilers, so we will be especially interested in parsers.

Two general approaches to parsing exist.

The first approach is *top-down*.

A parser is top-down if it "discovers" the parse tree corresponding to a token sequence by starting at the top of the tree (the start symbol), and then expanding the tree (via predictions) in a depth-first manner.

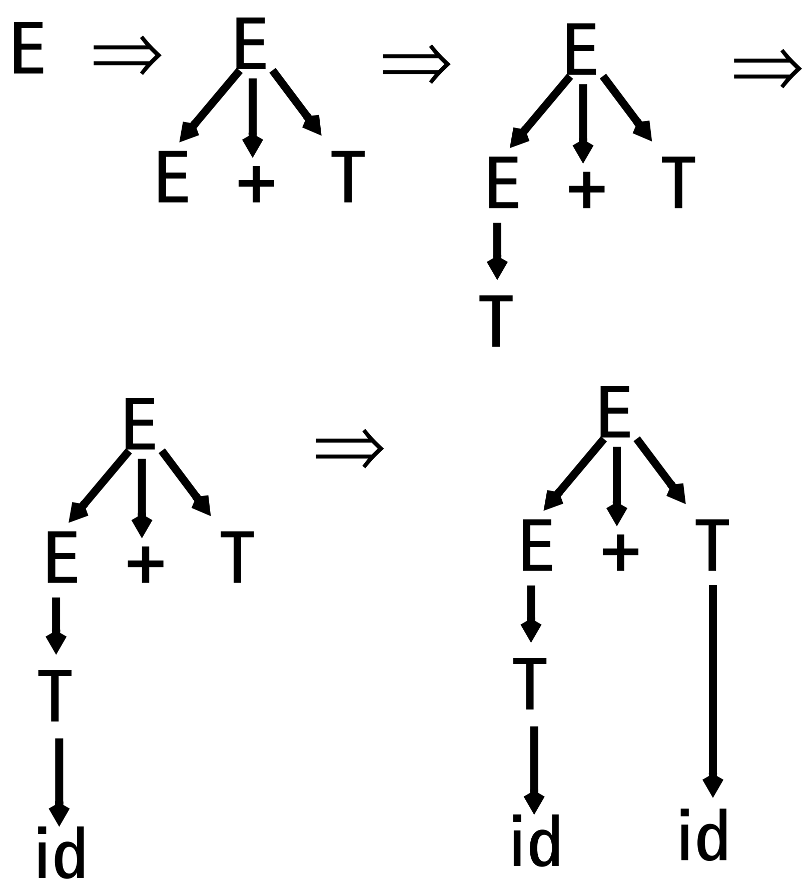
Top-down parsing techniques are *predictive* in nature because they always predict the production that is to be matched before matching actually begins.

Consider

E → E + T | T

T → T * id | id

To parse **id+id** in a top-down manner, a parser build a parse tree in the following steps:



A wide variety of parsing techniques take a different approach.

They belong to the class of *bottom-up* parsers.

As the name suggests, bottom-up parsers discover the structure of a parse tree by beginning at its bottom (at the leaves of the tree which are terminal symbols) and determining the productions used to generate the leaves.

Then the productions used to generate the immediate parents of the leaves are discovered.

The parser continues until it reaches the production used to expand the start symbol.

At this point the entire parse tree has been determined.

A bottom-up parse of **id₁+id₂** would follow the following steps: