CS 536

Introduction to Programming Languages and Compilers

Charles N. Fischer

Lecture 6

Optimizing Finite Automata

We can improve the DFA created by **`MakeDeterministic`**.

Sometimes a DFA will have more states than necessary. For every DFA there is a unique *smallest* equivalent DFA (fewest states possible).

Some DFA's contain *unreachable states* that cannot be reached from the start state.

Other DFA's may contain *dead states* that cannot reach any accepting state.

It is clear that neither unreachable states nor dead states can participate in scanning any valid token. We therefore eliminate all such states as part of our optimization process.

We optimize a DFA by *merging together* states we know to be equivalent.

For example, two accepting states that have no transitions at all out of them are equivalent.

Why? Because they behave exactly the same way—they accept the string read so far, but will accept no additional characters.

If two states, s_1 and s_2, are equivalent, then all transitions to s_2 can be replaced with transitions to s_1. In effect, the two states are merged together into one common state.

How do we decide what states to merge together?

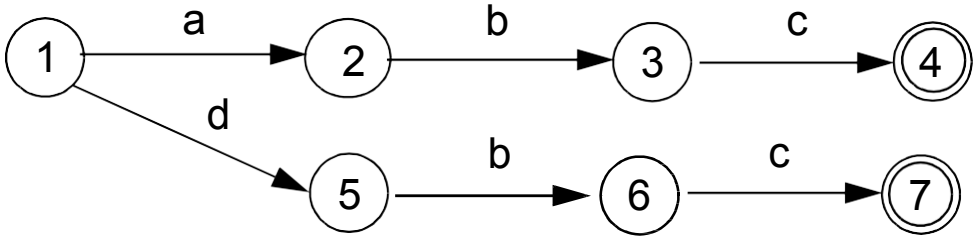
We take a *greedy* approach and try the most optimistic merger of states. By definition, accepting and non- accepting states are distinct, so we initially try to create only two states: one representing the merger of all accepting states and the other representing the merger of all non- accepting states.

This merger into only two states is almost certainly too optimistic. In particular, all the constituents of a merged state must agree on the same transition for each possible character. That is, for character c all the merged states must have no successor under c or they must all go to a single (possibly merged) state.

If all constituents of a merged state do not agree on the

transition to follow for some character, the merged state is *split* into two or more smaller states that do agree.

As an example, assume we start with the following automaton:



Initially we have a merged non-accepting state {1,2,3,5,6} and a merged accepting state {4,7}.

A merger is legal if and only if all constituent states agree on the same successor state for all characters. For example, states 3 and 6 would go to an accepting state given character c; states 1, 2, 5 would not, so a split must occur.

We will add an error state s_E to the original DFA that is the successor state under any illegal character. (Thus reaching s_E becomes equivalent to detecting an illegal token.) s_E is not a real state; rather it allows us to assume every state has a successor under every character. s_E is never merged with any real state.

Algorithm **`Split`**, shown below, splits merged states whose constituents do not agree on a common successor state for all characters. When **`Split`** terminates, we know that the states that remain merged are equivalent in that they always agree on common successors.

```
Split(FASet StateSet)
 { repeat
  for(each merged state S in StateSet)
    { Let S correspond to {s1,…,sn}
    for(each char c in Alphabet)
      { Let t1,…,tn be the successor
      states to s1,…,sn under c
      if(t1,…,tn do not all belong to
          the same merged state){
           Split S into two or more new
           statessuch that si and sj
           remainin the same merged
           state if and only if ti and tj
           are in the same merged state}
      }
 until no more splits are possible
}
```

Returning to our example, we initially have states {1,2,3,5,6} and {4,7}. Invoking **`Split`**, we first observe that states 3 and 6 have a common successor under c, and states 1, 2, and 5 have no successor under c (equivalently, have the error state s_E as a successor).

This forces a split, yielding {1,2,5}, {3,6} and {4,7}.

Now, for character b, states 2 and 5 would go to the merged state {3,6}, but state 1 would not, so another split occurs.

We now have: {1}, {2,5}, {3,6} and {4,7}.

At this point we are done, as all constituents of merged states agree on the same successor for each input symbol.

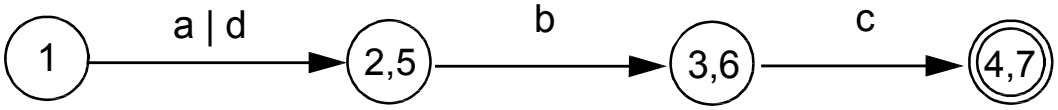
Once **`Split`** is executed, we are essentially done.

Transitions between merged states are the same as the transitions between states in the original DFA.

Thus, if there was a transition between state s_i and s_j under character c, there is now a transition under c from the merged state containing s_i to the merged state containing s_j. The start state is that merged state containing the original start state.

Accepting states are those merged states containing accepting states (recall that accepting and non- accepting states are never merged).

Returning to our example, the minimum state automaton we obtain is



Properties of Regular Expressions and Finite Automata

- Some token patterns *can't* be defined as regular expressions or finite automata. Consider the set of balanced brackets of the form [[[.]]].
  This set is defined formally as
  $\{ [^m]^m \mid m \geq 1 \}$.
  This set is *not* regular.
  No finite automaton that recognizes *exactly* this set can exist.
  Why? Consider the inputs [, [[, [[[, ...
  For two different counts (call them i and j) $[^i$ and $[^j$ must reach the same state of a given FA! (Why?)
  Once that happens, we know that if $[^i]^i$ is accepted (as it should be), the $[^j]^i$ will also be accepted (and that should not happen).

- $\overline{R} = V^* - R$ is regular if R is.
  Why?
  Build a finite automaton for R. Be careful to include transitions to an "error state" s_E for illegal characters.
  Now invert final and non- final states. What was previously accepted is now rejected, and what was rejected is now accepted. That is, $\overline{R}$ is accepted by the modified automaton.

- Not all subsets of a regular set are themselves regular. The regular expression $[^+]^+$ has a subset that isn't regular. (What is that subset?)

- Let R be a set of strings. Define R^{rev} as all strings in R, in reversed (backward) character order.
  Thus if R = {abc, def}

  then R^{rev} = {cba, fed}.

  If R is regular, then R^{rev} is too.
  Why? Build a finite automaton for R. Make sure the automaton has only one final state. Now *reverse* the direction of all transitions, and interchange the start and final states. What does the modified automation accept?

- If R_1 and R_2 are both regular, then $R_1 \cap R_2$ is also regular. We can show this two different ways:
  1. Build two finite automata, one for R1 and one for R2. Pair together states of the two automata to match R1 and R2 simultaneously. The paired-state automaton accepts only if both R1 and R2 would, so R1 ∩ R2 is matched.
  2. We can use the fact that R1 ∩ R2 is = $\overline{\overline{R_1} \cup \overline{R_2}}$ We already know union and complementation are regular.

Reading Assignment

- Read Chapter 4 of Crafting a Compiler

Context Free Grammars

A context- free grammar (CFG) is defined as:

- A finite terminal set V_t;
  these are the tokens produced by the scanner.
- A set of intermediate symbols, called non- terminals, V_n.
- A start symbol, a designated non-terminal, that starts all derivations.
- A set of productions (sometimes called rewriting rules) of the form
  $A \rightarrow X_1 \ldots X_m$
  X_1 to X_m may be any combination of terminals and non- terminals.
  If $m = 0$ we have $A \rightarrow \lambda$
  which is a valid production.

Example

Prog → { Stmts }

Stmts →Stmts ; Stmt

Stmts →Stmt

Stmt →id = Expr

Expr →id

Expr →Expr + id

Often more than one production shares the same left- hand side.

Rather than repeat the left hand side, an "or notation" is used:

Prog → { Stmts }

Stmts →Stmts ; Stmt

| Stmt

Stmt →id = Expr

Expr →id

| Expr + id

Derivations

Starting with the start symbol, non-terminals are rewritten using productions until only terminals remain.

Any terminal sequence that can be generated in this manner is syntactically valid.

If a terminal sequence can't be generated using the productions of the grammar it is invalid (has syntax errors).

The set of strings derivable from the start symbol is the *language* of the grammar (sometimes denoted L(G)).

For example, starting at **Prog** we generate a terminal sequence, by repeatedly applying productions:

Prog

{ Stmts }

{ Stmts ; Stmt }

{ Stmt ; Stmt }

{ id = Expr ; Stmt }

{ id = id ; Stmt }

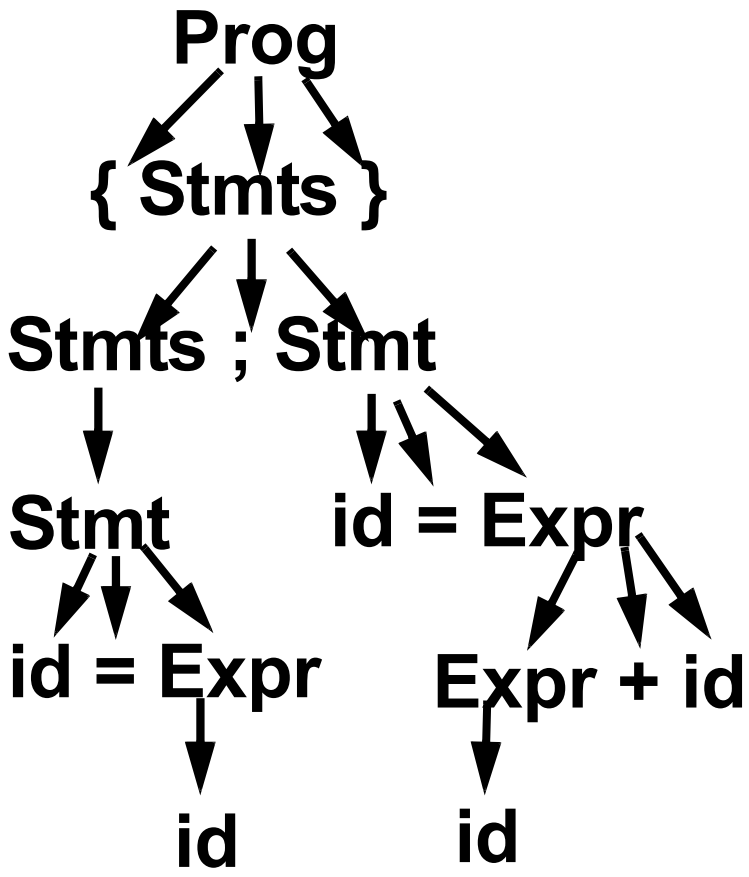
{ id = id ; id = Expr }

{ id = id ; id = Expr + id}

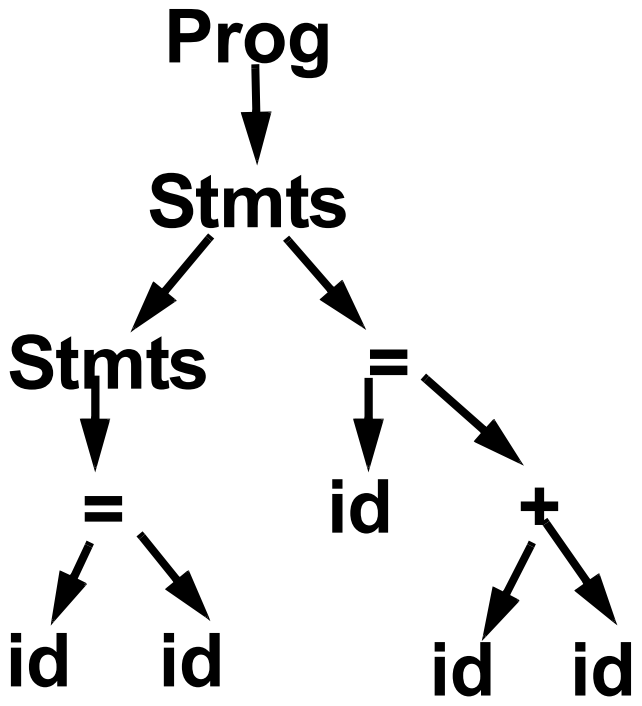
{ id = id ; id = id + id}

Parse Trees

To illustrate a derivation, we can draw a *derivation tree* (also called a *parse tree*):



An *abstract syntax tree* (AST) shows essential structure but eliminates unnecessary delimiters and intermediate symbols:



If $A \rightarrow \gamma$ is a production then
$\alpha A \beta \Rightarrow \alpha \gamma \beta$
where $\Rightarrow$ denotes a one step derivation (using production $A \rightarrow \gamma$).

We extend $\Rightarrow$ to $\Rightarrow^+$ (derives in one or more steps), and $\Rightarrow^*$ (derives in zero or more steps).

We can show our earlier derivation as

Prog $\Rightarrow$
{ Stmts } $\Rightarrow$
{ Stmts ; Stmt } $\Rightarrow$
{ Stmt ; Stmt } $\Rightarrow$
{ id = Expr ; Stmt } $\Rightarrow$
{ id = id ; Stmt } $\Rightarrow$
{ id = id ; id = Expr } $\Rightarrow$
{ id = id ; id = Expr + id} $\Rightarrow$
{ id = id ; id = id + id}

Prog $\Rightarrow^+$ **{ id = id ; id = id + id}**

When deriving a token sequence, if more than one non- terminal is present, we have a choice of which to expand next.

We must specify, at each step, which non- terminal is expanded, and what production is applied.

For simplicity we adopt a convention on what non- terminal is expanded at each step.

We can choose the leftmost possible non- terminal at each step.

A derivation that follows this rule is a *leftmost derivation*.

If we know a derivation is leftmost, we need only specify what productions are used; the choice of non- terminal is always fixed.

To denote derivations that are leftmost,

we use $\Rightarrow_L$, $\Rightarrow_L^+$, and $\Rightarrow_L^*$

The production sequence discovered by a large class of parsers (the top- down parsers) is a leftmost derivation, hence these parsers produce a *leftmost parse*.

Prog $\Rightarrow_L$

{ Stmts } $\Rightarrow_L$

{ Stmts ; Stmt } $\Rightarrow_L$

{ Stmt ; Stmt } $\Rightarrow_L$

{ id = Expr ; Stmt } $\Rightarrow_L$

{ id = id ; Stmt } $\Rightarrow_L$

{ id = id ; id = Expr } $\Rightarrow_L$

{ id = id ; id = Expr + id} $\Rightarrow_L$

{ id = id ; id = id + id}

Prog $\Rightarrow_L^+$ **{ id = id ; id = id + id}**

Rightmost Derivations

A *rightmost derivation* is an alternative to a leftmost derivation. Now the rightmost non- terminal is always expanded.

This derivation sequence may seem less intuitive given our normal left- to- right bias, but it corresponds to an important class of parsers (the bottom- up parsers, including CUP).

As a bottom- up parser discovers the productions used to derive a token sequence, it discovers a rightmost derivation, but in *reverse order*.

The last production applied in a rightmost derivation is the first that is discovered. The first production used, involving the start symbol, is discovered last.

The sequence of productions recognized by a bottom- up parser is a *rightmost parse*.

It is the exact reverse of the production sequence that represents a rightmost derivation.

For rightmost derivations, we use the notation $\Rightarrow_R$, $\Rightarrow_R^+$, and $\Rightarrow_R^*$

Prog $\Rightarrow_R$

{ Stmts } $\Rightarrow_R$

{ Stmts ; Stmt } $\Rightarrow_R$

{ Stmts ; id = Expr } $\Rightarrow_R$

{ Stmts ; id = Expr + id } $\Rightarrow_R$

{ Stmts ; id = id + id } $\Rightarrow_R$

{ Stmt ; id = id + id } $\Rightarrow_R$

{ id = Expr ; id = id + id } $\Rightarrow_R$

{ id = id ; id = id + id}

Prog $\Rightarrow^+$ **{ id = id ; id = id + id}**

You can derive the same set of tokens using leftmost and rightmost derivations; the only difference is the order in which productions are used.

Ambiguous Grammars

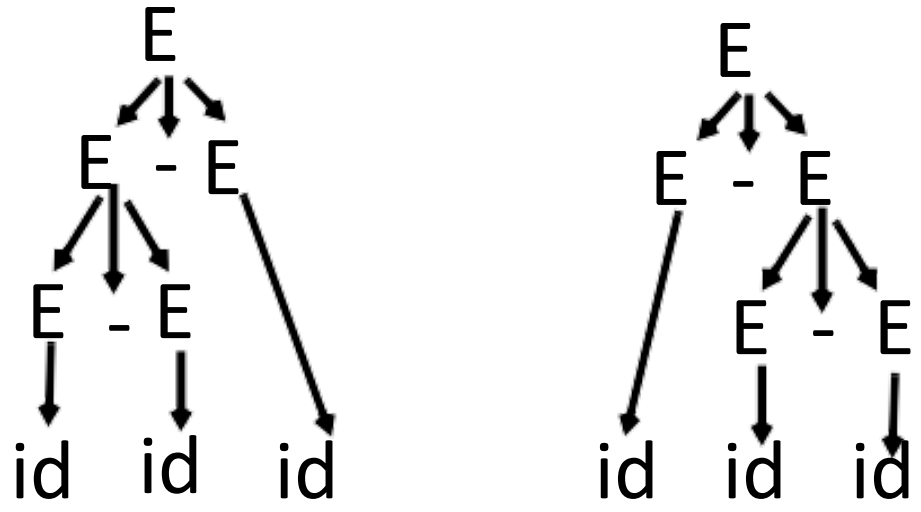
Some grammars allow more than one parse tree for the same token sequence. Such grammars are *ambiguous*. Because compilers use syntactic structure to drive translation, ambiguity is undesirable —it may lead to an unexpected translation.

Consider

E → E - E

| id

When parsing the input a- b- c (where a, b and c are scanned as identifiers) we can build the following two parse trees:



The effect is to parse a- b- c as either (a- b)- c or a- (b- c). These two groupings are certainly not equivalent.

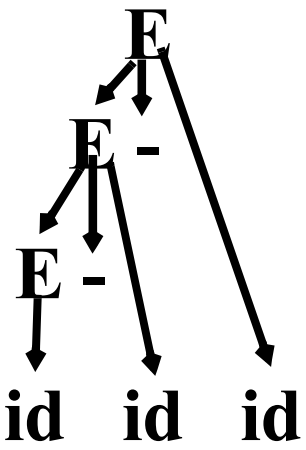
Ambiguous grammars are usually voided in building compilers; the tools we use, like Yacc and CUP, strongly prefer unambiguous grammars.

To correct this ambiguity, we use

E → E - id

| id

Now a- b- c can only be parsed as:



Operator Precedence

Most programming languages have *operator precedence* rules that state the order in which operators are applied (in the absence of explicit parentheses). Thus in C and Java and CSX, **`a +b*c`** means compute **`b*c`**, then add in **`a`**.

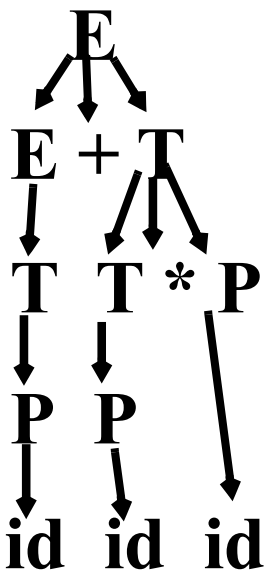
These operators precedence rules can be incorporated directly into a CFG.

Consider

E → E + T
| T
T → T * P
| P
P → id
| (E)

Does **`a+b*c`** mean **`(a+b)*c`** or **`a+(b*c)`**?

The grammar tells us! Look at the derivation tree:



The other grouping can't be obtained unless explicit parentheses are used.

(Why?)

Java CUP

Java CUP is a parser- generation tool, similar to Yacc.

CUP builds a Java parser for LALR(1) grammars from production rules and associated Java code fragments.

When a particular production is recognized, its associated code fragment is executed (typically to build an AST).

CUP generates a Java source file **`parser.java`**. It contains a class **`parser`**, with a method

```
Symbol parse()
```

The **`Symbol`** returned by the parser is associated with the grammar's start symbol and contains the AST for the whole source program.

The file **`sym.java`** is also built for use with a JLex- built scanner (so that both scanner and parser use the same token codes).

If an unrecovered syntax error occurs, **`Exception()`** is thrown by the parser.

CUP and Yacc accept exactly the same class of grammars—all LL(1) grammars, plus many useful non-LL(1) grammars.

CUP is called as

```
java java_cup.Main < file.cup
```

Java CUP Specifications

Java CUP specifications are of the form:

- Package and import specifications
- User code additions
- Terminal and non- terminal declarations
- A context- free grammar, augmented with Java code fragments

Package and Import Specifications

You define a package name as:

```
package name ;
```

You add imports to be used as:

```
import java_cup.runtime.*;
```

User Code Additions

You may define Java code to be included within the generated parser:

`action code {: /*java code */ :}`
This code is placed within the generated action class (which holds user- specified production actions).

`parser code {: /*java code */ :}`
This code is placed within the generated parser class.

`init with{: /*java code */ :}`
This code is used to initialize the generated parser.

`scan with{: /*java code */ :}`
This code is used to tell the generated parser how to get tokens from the scanner.

Terminal and Non-terminal Declarations

You define terminal symbols you will use as:

`terminal classname name`$_1$`, name`$_2$`, ...`

`classname` is a class used by the scanner for tokens (**`CSXToken`**, **`CSXIdentifierToken`**, etc.)

You define non- terminal symbols you will use as:

`non terminal classname name`$_1$`, name`$_2$`, ...`

`classname` is the class for the AST node associated with the non- terminal (**`stmtNode`**, **`exprNode`**, etc.)

Production Rules

Production rules are of the form

```
name ::= name1 name2 ... action ;
```

or

```
name ::= name1
action1  name2 ...
    |  name3 name4 ... action2
    |    ...
    ;
```

Names are the names of terminals or non- terminals, as declared earlier.

Actions are Java code fragments, of the form

```
{: /*java code */ :}
```

The Java object assocated with a symbol (a token or AST node) may be named by adding a **`:id`** suffix to a terminal or non- terminal in a rule.

RESULT names the left- hand side non- terminal.

The Java classes of the symbols are defined in the terminal and non- terminal declaration sections.

For example,

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

This corresponds to the production

prog → { stmts }

The left brace is named **l**; the stmts non- terminal is called **s**.

In the action code, a new **CSXLiteNode** is created and assigned to **prog**. It is constructed from the AST node associated with **s**. Its line and column

numbers are those given to the left brace, **`1`** (by the scanner).

To tell CUP what non- terminal to use as the start symbol (**`prog`** in our example), we use the directive:

```
start with prog;
```

Example

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

```
program →   { stmts }
stmts → stmt   stmts
      | λ
stmt → id   =   expr   ;
      | if   (   expr   )   stmt
expr   →   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); :}
;
```

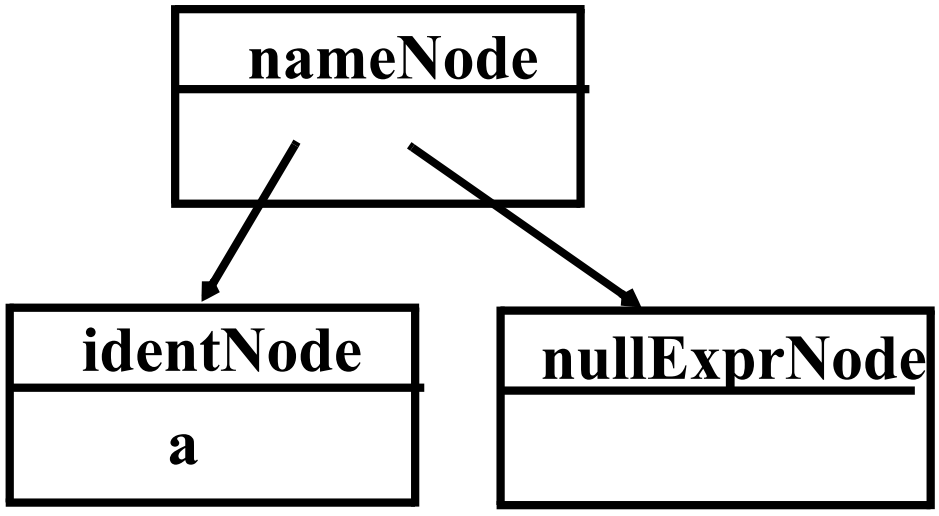
Let's parse

{ 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); :}
```

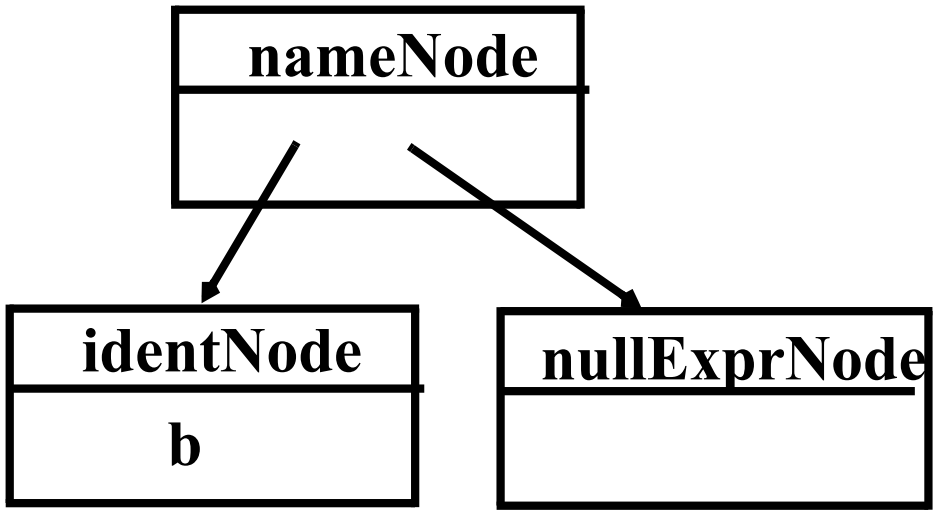
We build



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



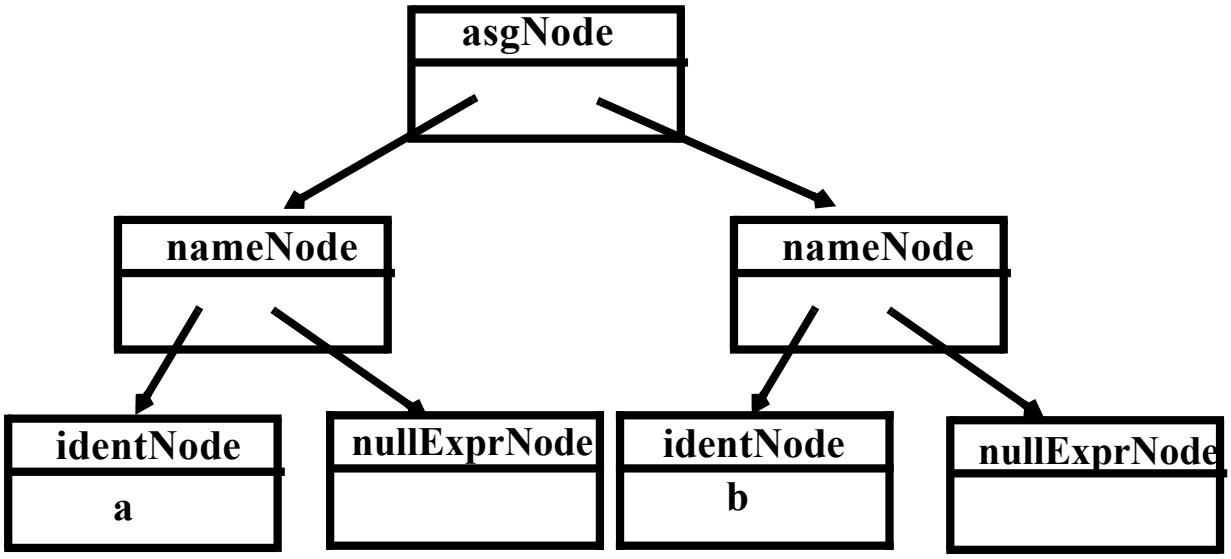
Then **b**'s subtree is recognized as an exp:

```
| 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

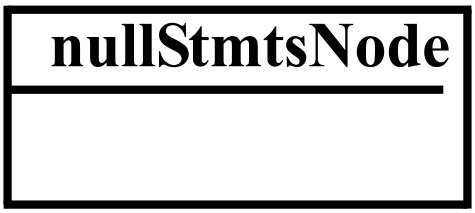


The stmts → λ production is matched (indicating that there are no more statements in the program).

CUP matches

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

and we build



Next,

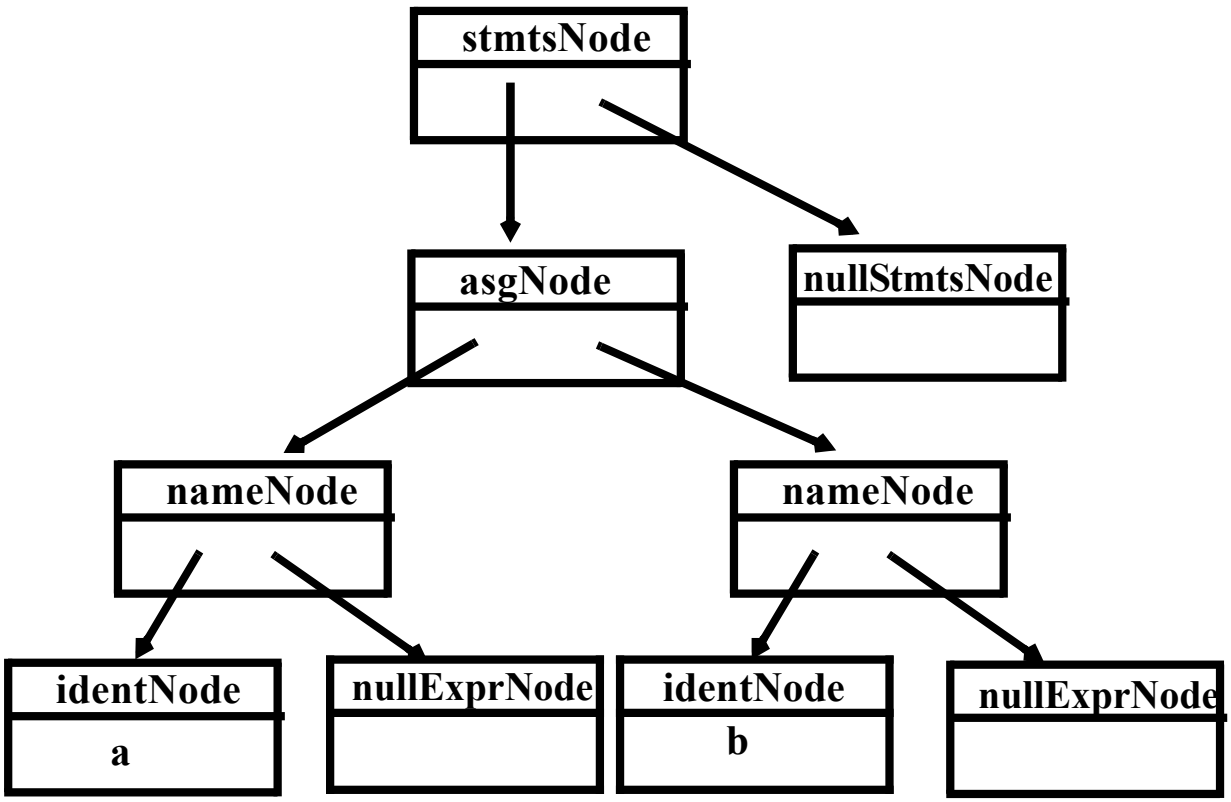
stmts → stmt stmts

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 → { 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