

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 name1, name2, ...
```

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 name1, name2, ...
```

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 name₂ ...

action₁

| 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 associated 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:1 stmts:s RBRACE  
{ : RESULT =  
    new csxLiteNode(s,  
        1.linenum, 1.colnum); : }
```

This corresponds to the production

prog → { **stmts** }

The left brace is named **1**; 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

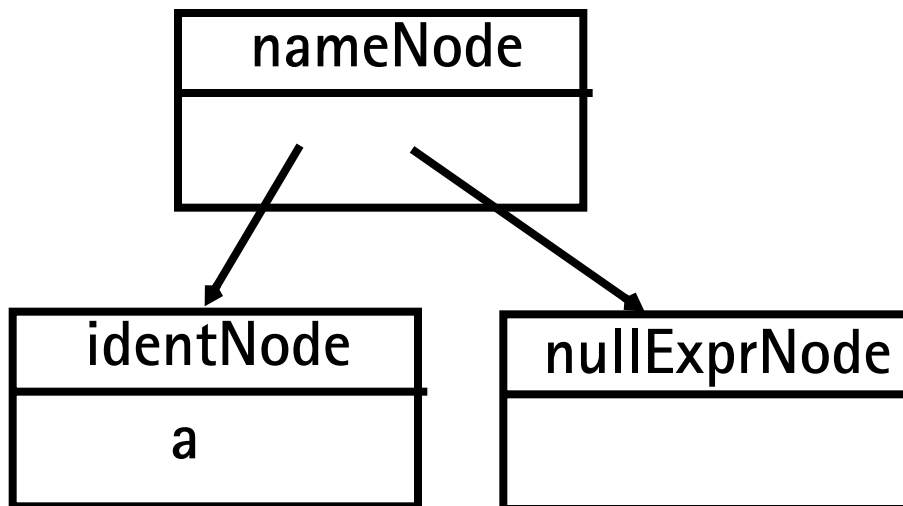
{ a = b ; }

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); :}
```

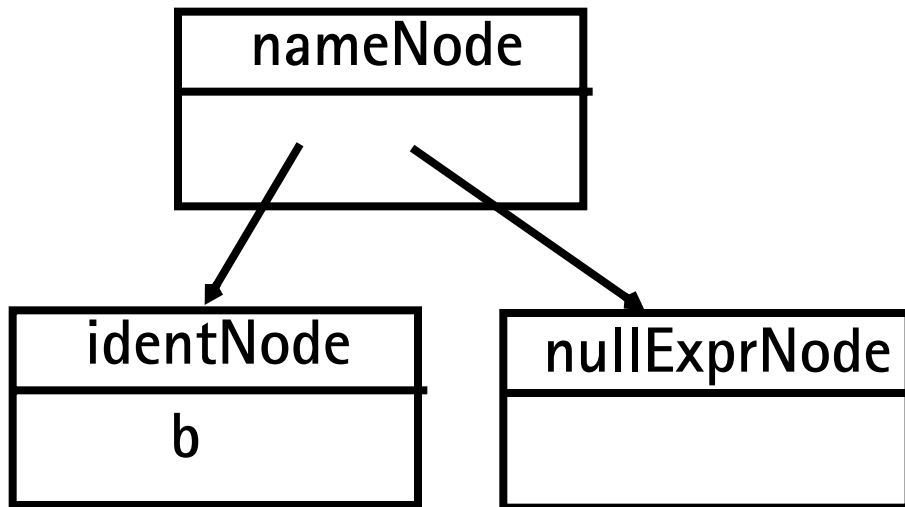
We build



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



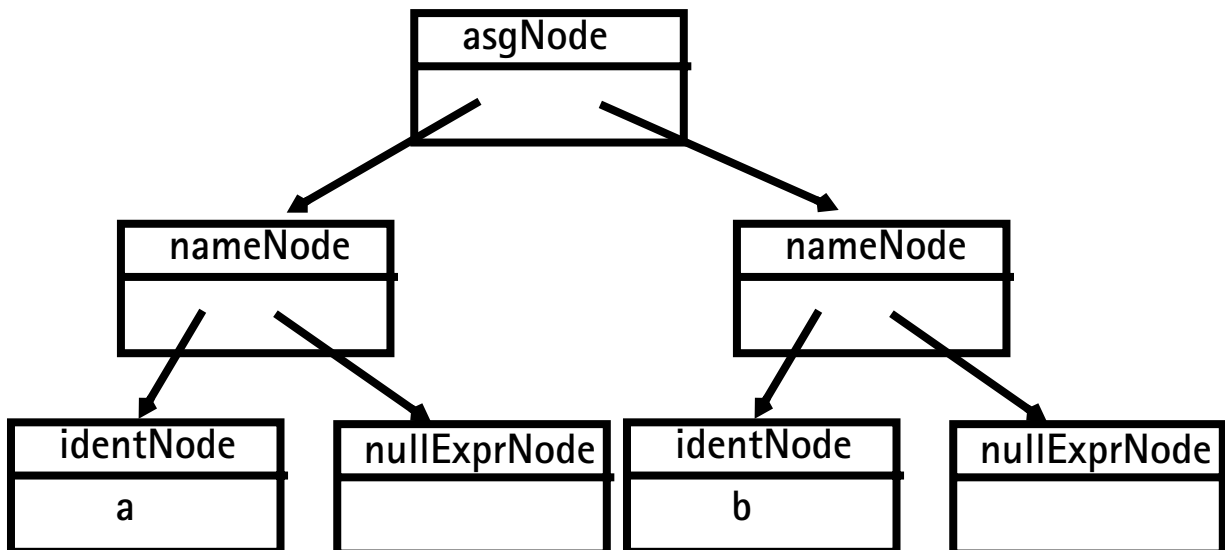
Then **b**'s subtree is recognized as an **exp**:

```
| 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



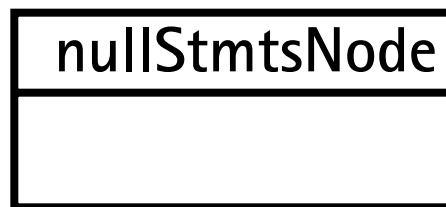
The **stmts** $\rightarrow \lambda$ production is matched (indicating that there are no more statements in the program).

CUP matches

```
stmts ::=
```

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

and we build



Next,

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

is matched using

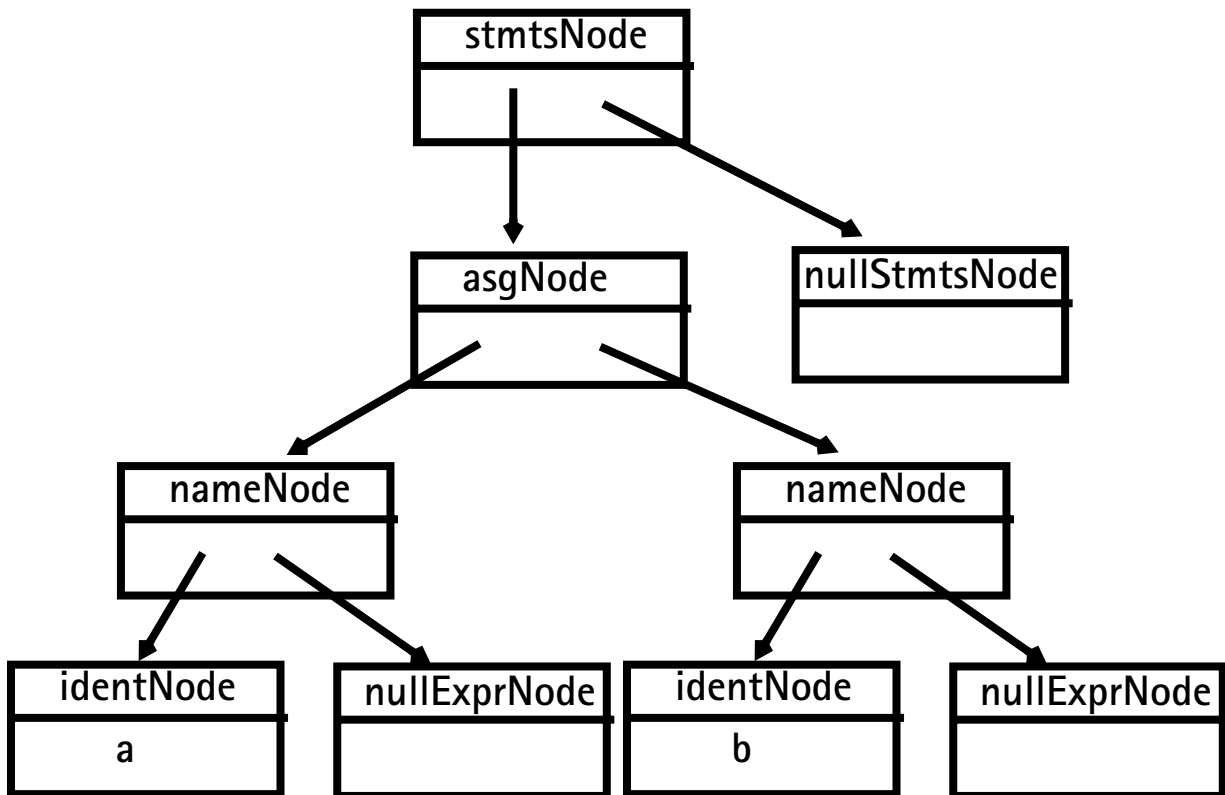
```
stmts ::= stmt:s1 stmts:s2
```

```
  { : RESULT=
```

```
    new stmtsNode(s1,s2,  
                  s1.linenum,s1.colnum);
```

```
  : }
```

This builds



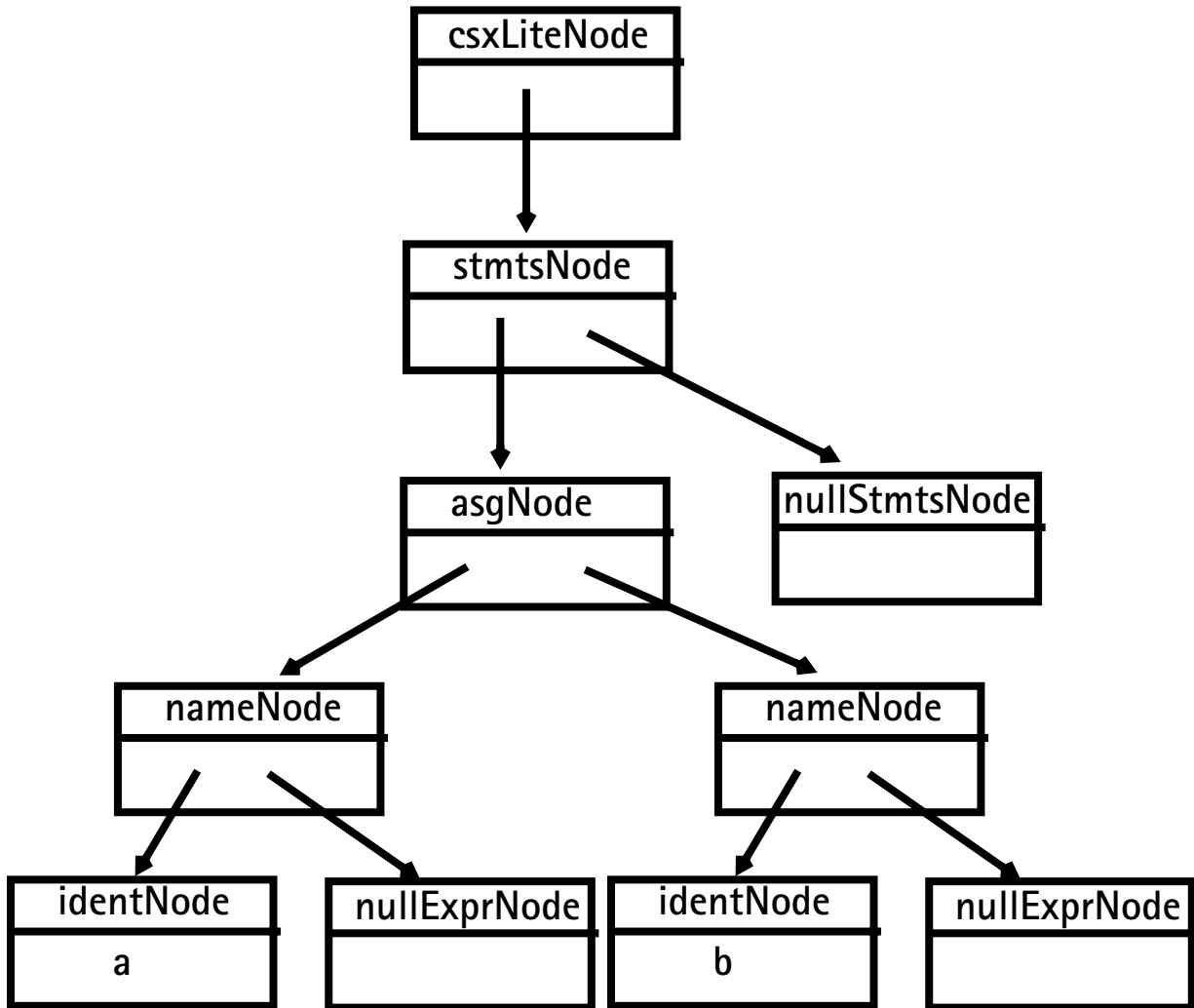
As the last step of the parse, the parser matches

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

using the CUP rule

```
prog ::= LBRACE:l stmts:s RBRACE
{ : RESULT=
    new csxLiteNode(s,
        1.linenum, 1.colnum); :}
;
```

The final AST returned 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

$S \rightarrow A B$

$\mid x$

$B \rightarrow b$

$A \rightarrow a A$

$C \rightarrow d$

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