CS 536

Introduction to Programming Languages and Compilers

Charles N. Fischer

Lecture 7

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 $\textbf{name}_1$**,** $\textbf{name}_2$**, ...**

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 $\textbf{name}_1$**,** $\textbf{name}_2$**, ...**

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 ::= name$_1$ name$_2$ ... 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 $\rightarrow$ { 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, **l** (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

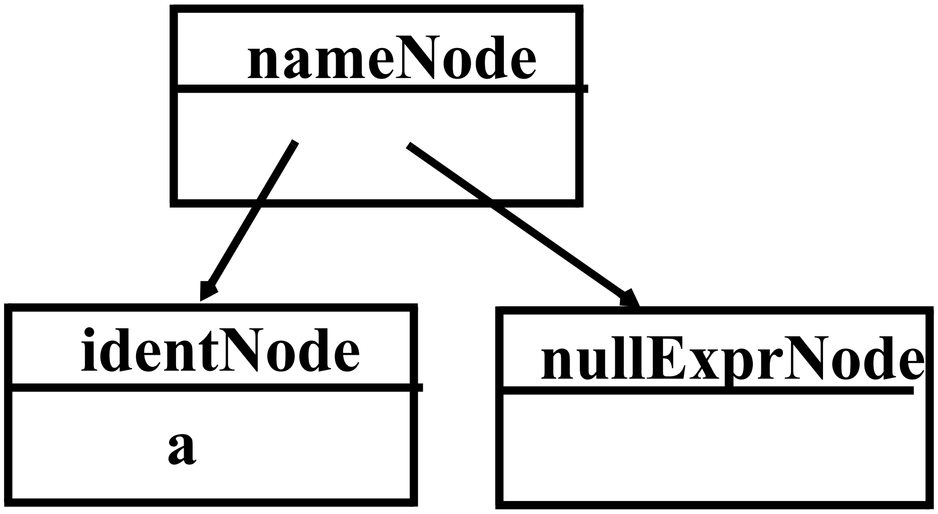
```
 { a = b ; }
```

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

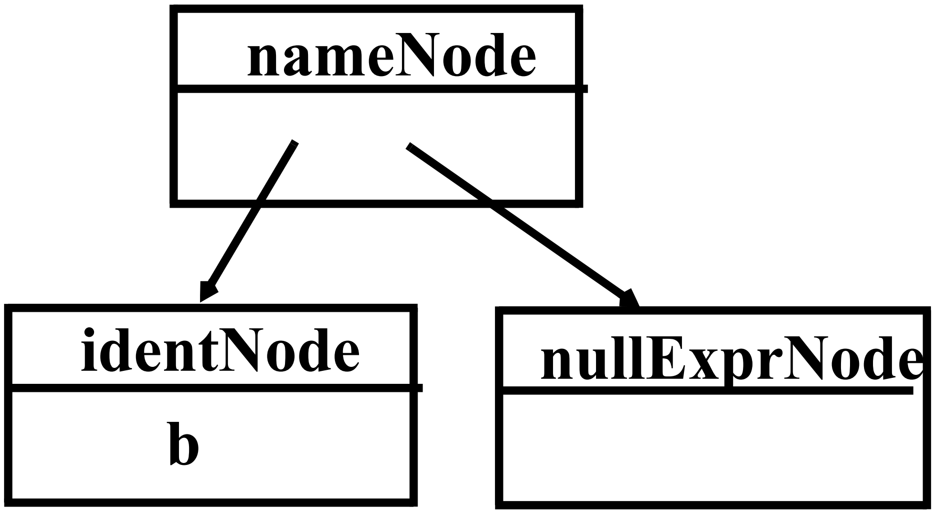


Next, **b** is parsed using

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

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

We build



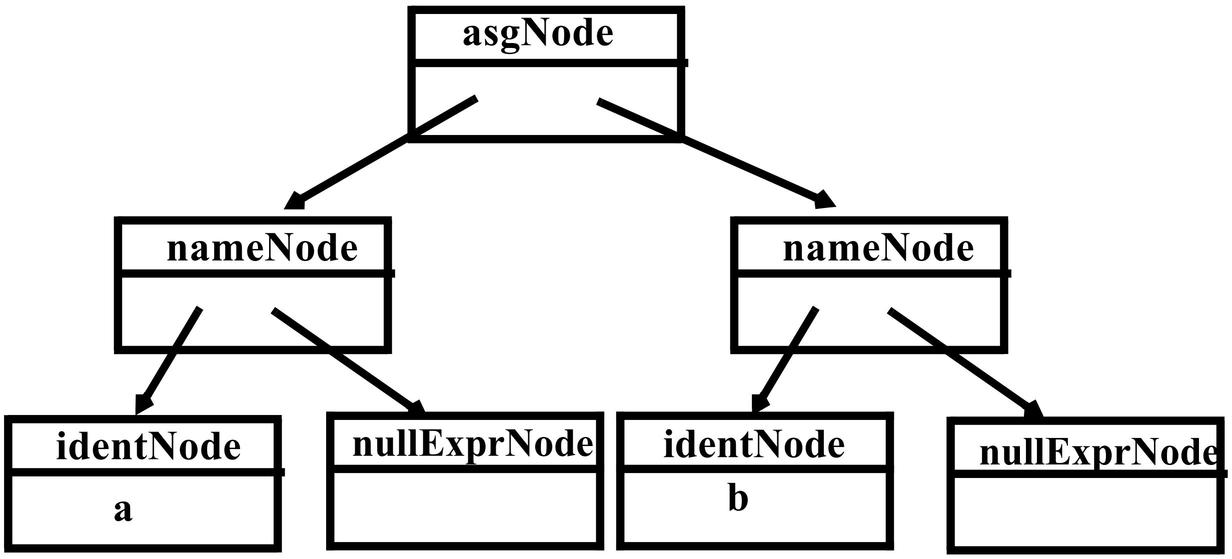
Then **`b`**’s subtree is recognized as an exp:

```
| 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

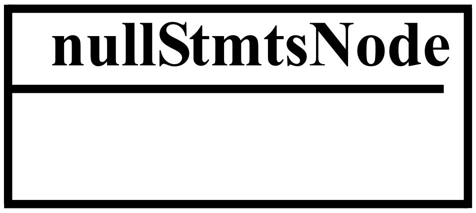


The stmts $\rightarrow$ λ production is matched (indicating that there are no more statements in the program).

CUP matches

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

and we build



Next,

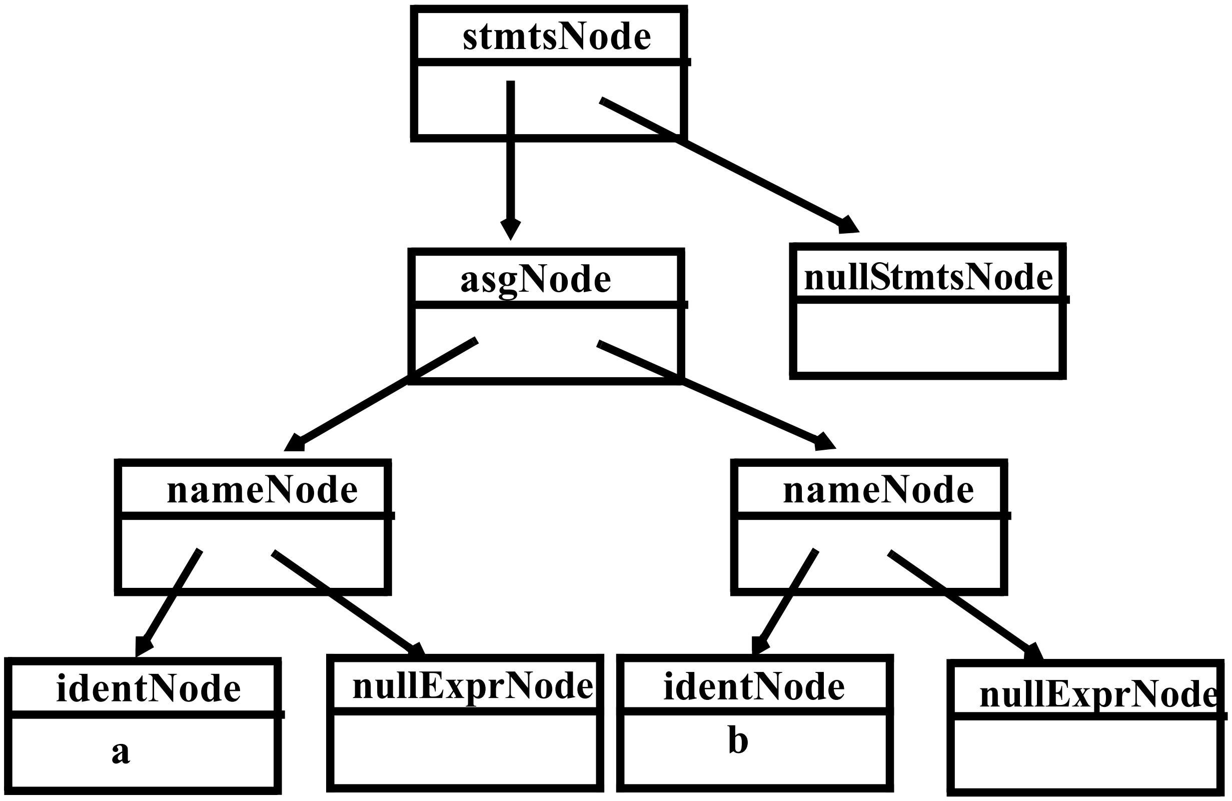
stmts $\rightarrow$ stmt   stmts

is matched using

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

 :}
```

This builds



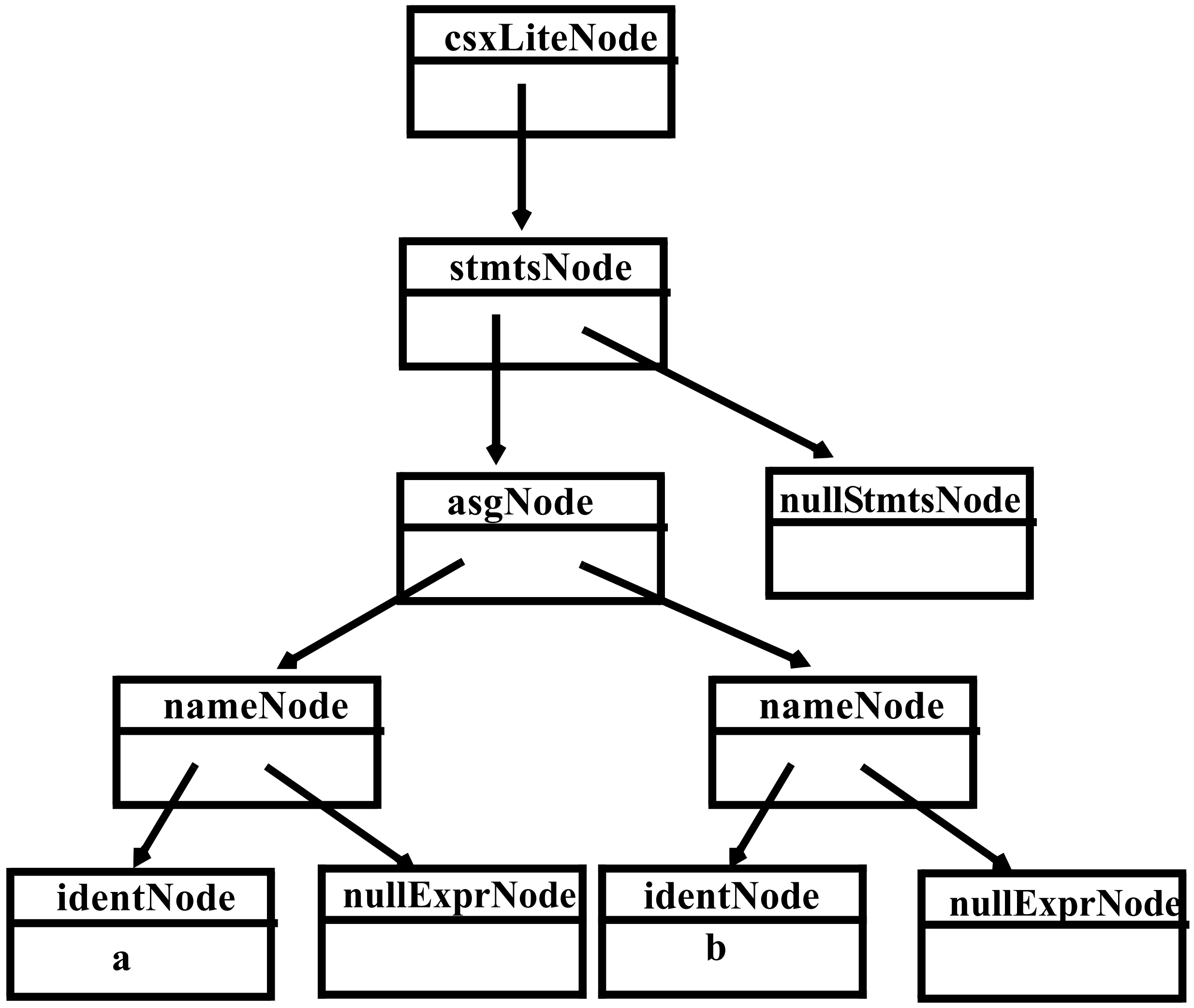
As the last step of the parse, the parser matches

program → { 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

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

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 → λ
        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

| λ

C $\rightarrow$ c

| λ

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.

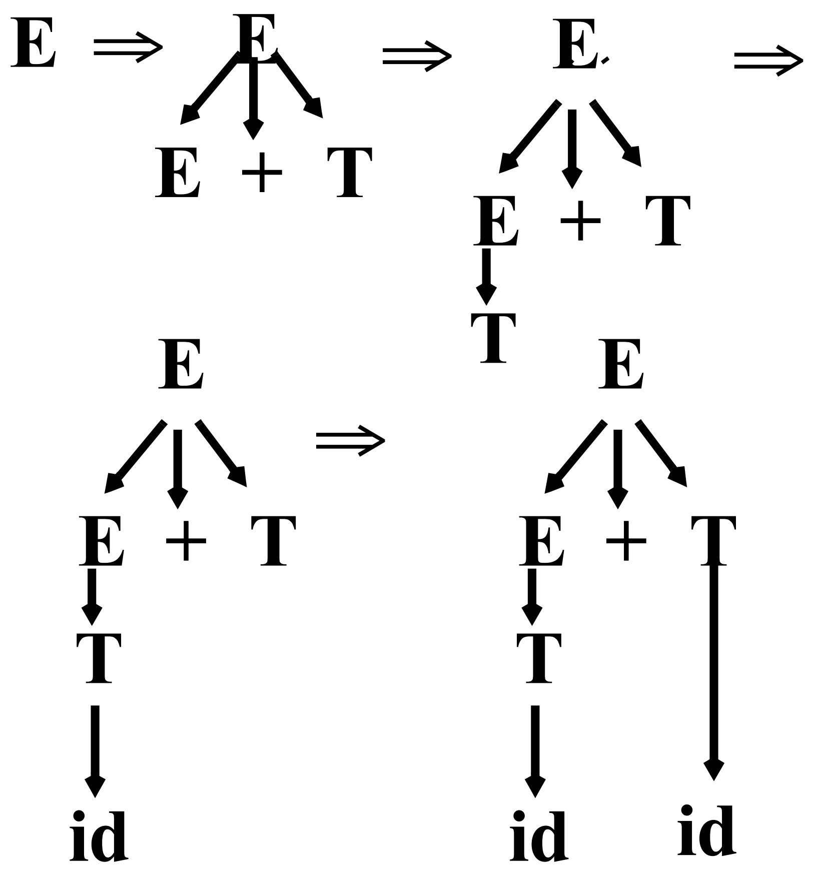
Top- down parsing techniques are *predictive* in nature because they always predict the production that is to be matched before matching actually begins.

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 id_1+id_2 would follow the following steps:

$$\begin{tikzcd}
T \arrow[d] \arrow[r, phantom, "\Rightarrow"] & E \arrow[d] \arrow[r, phantom, "\Rightarrow"] & {} \\
\mathbf{id_1} & T \arrow[d] & \\
& \mathbf{id_1} &
\end{tikzcd}$$

$$\begin{tikzcd}
T \arrow[d] \arrow[rr, phantom, "\Rightarrow"] & & E \arrow[dl] \arrow[d] \arrow[dr] & \\
\mathbf{id_2} & E \arrow[d] & + & T \arrow[dd] \\
& T \arrow[d] & & \\
& \mathbf{id_1} & & \mathbf{id_2}
\end{tikzcd}$$

A Simple Top-Down Parser

We'll build a rudimentary top-down parser that simply tries each possible expansion of a non-terminal, in order of production definition.

If an expansion leads to a token sequence that doesn't match the current token being parsed, we *backup* and try the next possible production choice.

We stop when all the input tokens are correctly matched or when all possible production choices have been tried.

Example

Given the productions

$$
\begin{aligned}
\mathbf{S} \rightarrow\ & \mathbf{a} \\
\mid\ & \mathbf{(\ S\ )}
\end{aligned}
$$

we try a, then (a), then ((a)), etc.

Let’s next try an additional alternative:

$$
\begin{aligned}
\mathbf{S} \rightarrow\ & \mathbf{a} \\
\mid\ & \mathbf{(\ S\ )} \\
\mid\ & \mathbf{(\ S\ ]}
\end{aligned}
$$

Let’s try to parse a, then (a], then ((a]], etc.

We’ll count the number of productions we try for each input.

- For input = a
  We try **S → a** which works.
  Matches needed = 1
- For input = (a]
  We try **S → a** which fails.
  We next try **S → (S)**.
  We expand the inner S three different ways; all fail.
  Finally, we try **S → (S]**.
  The inner S expands to a, which works.
  Total matches tried =
  1 + (1+3)+(1+1)= 7.
- For input = ((a]]
  We try **S → a** which fails.
  We next try **S → (S)**.
  We match the inner S to (a] using 7 steps, then fail to match the last].
  Finally, we try **S → (S]**.
  We match the inner S to (a] using 7

steps, then match the last].
Total matches tried =
1 + (1+7)+(1+7)= 17.

- For input = (((a]]]
  We try **S → a** which fails.
  We next try **S → (S)**.
  We match the inner S to ((a]] using 17 steps, then fail to match the last].
  Finally, we try **S → (S]**.
  We match the inner S to ((a]] using 17 steps, then match the last].
  Total matches tried =
  1 + (1+17) + (1+17) = 37.

Adding one extra (...] pair *doubles* the number of matches we need to do the parse.

In fact to parse $(^i a]^i$ takes $5*2^i - 3$ matches. This is *exponential* growth!

With a more effective dynamic programming approach, in which results of intermediate parsing steps are cached, we can reduce the
number of matches needed to n^3 for an input with n tokens.

Is this acceptable?

No!

Typical source programs have at

least 1000 tokens, and $1000^3 = 10^9$ is a lot of steps, even for a fast modern computer.

The solution?

—Smarter selection in the choice of productions we try.

Reading Assignment

Read Chapter 5 of Crafting a Compiler.

Prediction

We want to avoid trying productions that can’t possibly work.

For example, if the current token to be parsed is an identifier, it is useless to try a production that begins with an integer literal.

Before we try a production, we’ll consider the set of terminals it might initially produce. If the current token is in this set, we’ll try the production.

If it isn’t, there is no way the production being considered could be part of the parse, so we’ll ignore it.

A *predict function* tells us the set of tokens that might be initially generated from any production.

For $A \to X_1 \ldots X_n$, Predict($A \to X_1 \ldots X_n$) = Set of all initial (first) tokens derivable from $A \to X_1 \ldots X_n$

$= \{a \text{ in } V_t \mid A \to X_1 \ldots X_n \Rightarrow^* a \ldots\}$

For example, given

Stmt → Label id = Expr ;
| Label if Expr then Stmt ;
| Label read (IdList) ;
| Label id (Args) ;
Label → intlit :
| λ

Production	Predict Set
Stmt → Label id = Expr ;	**{id, intlit}**
Stmt → Label if Expr then Stmt ;	**{if, intlit}**
Stmt → Label read (IdList) ;	**{read, intlit}**
Stmt → Label id (Args) ;	**{id, intlit}**

We now will match a production p only if the next unmatched token is in p's predict set. We'll avoid trying productions that clearly won't work, so parsing will be faster.

But what is the predict set of a λ- production?

It can't be what's generated by λ (which is nothing!), so we'll define it as the tokens that can *follow* the use of a λ- production.

That is, Predict($A \rightarrow \lambda$) = Follow(A)

where (by definition)

Follow(A) = $\{a \text{ in } V_t \mid S \Rightarrow^+ \ldots Aa \ldots\}$

In our example,
Follow(Label $\rightarrow \lambda$) ={ id, if, read }

(since these terminals can immediately follow uses of Label in the given productions).

Now let’s parse

id (intlit) ;

Our start symbol is Stmt and the initial token is id.

id can predict

Stmt → Label id = Expr ;

id then predicts Label → λ

The id is matched, but “(“ doesn’t match “=” so we *backup* and try a different production for Stmt.

id also predicts

Stmt → Label id (Args) ;

Again, Label → λ is predicted and used, and the input tokens can match the rest of the remaining production.

We had only one misprediction, which is better than before.

Now we’ll rewrite the productions a bit to make predictions easier.

We remove the Label prefix from all the statement productions (now intlit won't predict all four productions).

We now have

Stmt → Label BasicStmt

BasicStmt → id = Expr ;

| if Expr then Stmt ;

| read (IdList) ;

| id (Args) ;

Label → intlit :

| λ

Now id predicts two different **BasicStmt** productions. If we rewrite these two productions into

BasicStmt → id StmtSuffix

StmtSuffix → = Expr ;

| (Args) ;

we no longer have any doubt over which production id predicts.

We now have

Production	Predict Set
Stmt → Label BasicStmt	**Not needed!**
BasicStmt → id StmtSuffix	**{id}**
BasicStmt → if Expr then Stmt ;	**{if}**
BasicStmt → read (IdList) ;	**{read}**
StmtSuffix → (Args) ;	**{ (}**
StmtSuffix → = Expr ;	**{ = }**
Label → intlit :	**{intlit}**
Label → λ	**{if, id, read}**

This grammar generates the same statements as our original grammar did, but now prediction never fails!

Whenever we must decide what production to use, the predict sets for productions with the same lefthand side are always disjoint.

Any input token will predict a unique production or no production at all (indicating a syntax error).

If we never mispredict a production, we never backup, so parsing will be fast and absolutely accurate!

LL(1) Grammars

A context-free grammar whose Predict sets are always disjoint (for the same non-terminal) is said to be *LL(1)*.

LL(1) grammars are ideally suited for top-down parsing because it is always possible to correctly predict the expansion of any non-terminal. No backup is ever needed.

Formally, let

First($X_1 \ldots X_n$) =

$\{a \text{ in } V_t \mid A \rightarrow X_1 \ldots X_n \Rightarrow^* a\ldots\}$

Follow(A) = $\{a \text{ in } V_t \mid S \Rightarrow^+ \ldots Aa\ldots\}$

Predict($A \rightarrow X_1 \ldots X_n$) =

If $X_1 \ldots X_n \Rightarrow^* \lambda$

Then First($X_1 \ldots X_n$) U Follow(A)

Else First($X_1 \ldots X_n$)

If some CFG, G, has the property that for all pairs of distinct productions with the same lefthand side,
$A \rightarrow X_1 \ldots X_n$ and $A \rightarrow Y_1 \ldots Y_m$
it is the case that

Predict($A \rightarrow X_1 \ldots X_n$) $\cap$

Predict($A \rightarrow Y_1 \ldots Y_m$) = ϕ

then G is LL(1).

LL(1) grammars are easy to parse in a top- down manner since predictions are always correct.

Example

Production	Predict Set
$S \rightarrow A\ a$	{b,d,a}
$A \rightarrow B\ D$	{b, d, a}
$B \rightarrow b$	{ b }
$B \rightarrow \lambda$	{d, a}
$D \rightarrow d$	{ d }
$D \rightarrow \lambda$	{ a }

Since the predict sets of both B productions and both D productions are disjoint, this grammar is LL(1).