How do LL(1) Parsers Build Syntax Trees?

So far our LL(1) parser has acted like a recognizer. It verifies that input token are syntactically correct, but it produces no output.

Building complete (concrete) parse trees automatically is fairly easy.

As tokens and non-terminals are matched, they are pushed onto a second stack, the *semantic stack*.

At the end of each production, an action routine pops off n items from the semantic stack (where n is the length of the production's righthand side). It then builds a syntax tree whose root is the lefthand side, and

whose children are the n items just popped off.

For example, for production

Stmt → id = Expr ;

the parser would include an action symbol after the ";" whose actions are:

P4 = pop(); // Semicolon token

P3 = pop(): // Syntax tree for Expr

P2 = pop(); // Assignment token

P1 = pop(); // Identifier token

Push(new StmtNode(P1,P2,P3,P4));

Creating Abstract Syntax Trees

Recall that we prefer that parsers generate abstract syntax trees, since they are simpler and more concise.

Since a parser generator can't know what tree structure we want to keep, we must allow the user to define "custom" action code, just as Java CUP does.

We allow users to include "code snippets" in Java or C. We also allow labels on symbols so that we can refer to the tokens and tress we wish to access. Our production and action code will now look like this:

Stmt → id:i = Expr:e ;

```
{: RESULT = new StmtNode(i,e); :}
```


How do We Make Grammars LL(1)?

Not all grammars are LL(1); sometimes we need to modify a grammar's productions to create the disjoint Predict sets LL1) requires.

There are two common problems in grammars that make unique prediction difficult or impossible:

1. Common prefixes.
   Two or more productions with the same lefthand side begin with the same symbol(s).
   For example,

   Stmt → id = Expr ;
   Stmt → id (Args) ;

2. Left-Recursion

A production of the form

A $\rightarrow$ **A ...**

is said to be left-recursive.

When a left-recursive production is used, a non-terminal is immediately replaced by itself (with additional symbols following).

Any grammar with a left-recursive production can *never* be LL(1).

Why?

Assume a non-terminal A reaches the top of the parse stack, with CT as the current token. The LL(1) parse table entry, T[A][CT], predicts **A** $\rightarrow$ **A ...**

We expand A again, and T[A][CT], so we predict **A** $\rightarrow$ **A ...** again. We are in an infinite prediction loop!

Eliminating Common Prefixes

Assume we have two of more productions with the same lefthand side and a common prefix on their righthand sides:

A $\rightarrow \alpha\,\beta \mid \alpha\,\gamma \mid \ldots \mid \alpha\,\delta$

We create a new non-terminal, **X**.

We then rewrite the above productions into:

A $\rightarrow \alpha$**X** **X** $\rightarrow \beta \mid \gamma \mid \ldots \mid \delta$

For example,

Stmt $\rightarrow$ **id = Expr ;**
Stmt $\rightarrow$ **id (Args) ;**

becomes

Stmt $\rightarrow$ **id StmtSuffix**
StmtSuffix $\rightarrow$ **= Expr ;**
StmtSuffix $\rightarrow$ **(Args) ;**

Eliminating Left Recursion

Assume we have a non-terminal that is left recursive:

A $\rightarrow$ **A**α **A** $\rightarrow \beta \mid \gamma \mid \ldots \mid \delta$

To eliminate the left recursion, we create two new non-terminals, **N** and **T**.

We then rewrite the above productions into:

A $\rightarrow$ **N T** **N** $\rightarrow \beta \mid \gamma \mid \ldots \mid \delta$

T $\rightarrow \alpha$ **T** $\mid \lambda$

For example,

Expr $\rightarrow$ **Expr + id**
Expr $\rightarrow$ **id**

becomes

Expr $\rightarrow$ **N T**
N $\rightarrow$ **id**
T $\rightarrow$ **+ id T** $\mid \lambda$

This simplifies to:

Expr $\rightarrow$ **id T**
T $\rightarrow$ **+ id T** $\mid \lambda$

Reading Assignment

Get Chapter 6 of Crafting a Compiler featuring Java. Read Sections 6.1 to 6.5.1.

(Available from DoIt Tech Store)

How does JavaCup Work?

The main limitation of LL(1) parsing is that it must predict the correct production to use when it first starts to match the production's righthand side.

An improvement to this approach is the LALR(1) parsing method that is used in JavaCUP (and Yacc and Bison too).

The LALR(1) parser is bottom-up in approach. It tracks the portion of a righthand side already matched as tokens are scanned. It may not know immediately which is the correct production to choose, so it tracks *sets* of possible matching productions.

Configurations

We'll use the notation

$\mathbf{X} \rightarrow \mathbf{A\ B \cdot C\ D}$

to represent the fact that we are trying to match the production $\mathbf{X} \rightarrow \mathbf{A\ B \cdot C\ D}$ with **A** and **B** matched so far.

A production with a "•" somewhere in its righthand side is called a *configuration*.

Our goal is to reach a configuration with the "dot" at the extreme right:

$\mathbf{X} \rightarrow \mathbf{A\ B\ C\ D \cdot}$

This indicates that an entire production has just been matched.

Since we may not know which production will eventually be fully

matched, we may need to track a *configuration set*. A configuration set is sometimes called a *state*.

When we predict a production, we place the "dot" at the beginning of a production:

$\mathbf{X} \rightarrow \mathbf{\cdot\ A\ B\ C\ D}$

This indicates that the production may possibly be matched, but no symbols have actually yet been matched.

We may predict a λ-production:

$\mathbf{X} \rightarrow \lambda \cdot$

When a λ-production is predicted, it is immediately matched, since λ can be matched at any time.

Starting the Parse

At the start of the parse, we know some production with the start symbol must be used initially. We don't yet know which one, so we predict them *all*:

S $\rightarrow$ $\cdot$ **A B C D**

S $\rightarrow$ $\cdot$ **e F g**

S $\rightarrow$ $\cdot$ **h I**

...

Closure

When we encounter a configuration with the dot to the left of a non-terminal, we know we need to try to match that non-terminal.

Thus in

X $\rightarrow$ $\cdot$ **A B C D**

we need to match some production with A as its left hand side.

Which production?

We don't know, so we predict *all* possibilities:

A $\rightarrow$ $\cdot$ **P Q R**

A $\rightarrow$ $\cdot$ **s T**

...

The newly added configurations may predict other non-terminals, forcing

additional productions to be included. We continue this process until no additional configurations can be added.

This process is called *closure* (of the configuration set).

Here is the closure algorithm:

```
ConfigSet Closure(ConfigSet C){
  repeat
     if (X → α •B δ is in C &&
            B is a non-terminal)
          Add all configurations of
           the form B → •γ to C)
  until (no more configurations
              can be added);
  return C;
}
```


Example of Closure

Assume we have the following grammar:

S $\rightarrow$ **A b**

A $\rightarrow$ **C D**

C $\rightarrow$ **D**

C $\rightarrow$ **c**

D $\rightarrow$ **d**

To compute Closure(**S** $\rightarrow$ $\cdot$ **A b**)

we first include all productions that rewrite A:

A $\rightarrow$ $\cdot$ **C D**

Now **C** productions are included:

C $\rightarrow$ $\cdot$ **D**

C $\rightarrow$ $\cdot$ **c**

Finally, the D production is added:

D $\rightarrow$ $\cdot$ **d**

The complete configuration set is:

S $\rightarrow$ $\cdot$ **A b**

A $\rightarrow$ $\cdot$ **C D**

C $\rightarrow$ $\cdot$ **D**

C $\rightarrow$ $\cdot$ **c**

D $\rightarrow$ $\cdot$ **d**

This set tells us that if we want to match an **A**, we will need to match a **C**, and this is done by matching a **c** or **d** token.

Shift Operations

When we match a symbol (a terminal or non-terminal), we *shift* the "dot" past the symbol just matched. Configurations that don't have a dot to the left of the matched symbol are deleted (since they didn't correctly anticipate the matched symbol).

The **GoTo** function computes an updated configuration set after a symbol is shifted:

```
ConfigSet GoTo(ConfigSet C,
               Symbol X){
 B=φ;
 for each configuration f in C{
  if (f is of the form A → α•X δ)
       Add A → α X •δ to B;
 return Closure(B);
}
```


For example, if **c** is

S $\rightarrow$ $\cdot$ **A b**

A $\rightarrow$ $\cdot$ **C D**

C $\rightarrow$ $\cdot$ **D**

C $\rightarrow$ $\cdot$ **c**

D $\rightarrow$ $\cdot$ **d**

and **X** is **C**, then **GoTo** returns

A $\rightarrow$ **C** $\cdot$ **D**

D $\rightarrow$ $\cdot$ **d**

Reduce Actions

When the dot in a configuration reaches the rightmost position, we have matched an entire righthand side. We are ready to replace the righthand side symbols with the lefthand side of the production. The lefthand side symbol can now be considered matched.

If a configuration set can shift a token and also reduce a production, we have a potential *shift/reduce error*.

If we can reduce more than one production, we have a potential *reduce/reduce error*.

How do we decide whether to do a shift or reduce? How do we choose among more than one reduction?

We examine the next token to see if it is consistent with the potential reduce actions.

The simplest way to do this is to use Follow sets, as we did in LL(1) parsing.

If we have a configuration

$\mathbf{A} \rightarrow \alpha \cdot$

we will reduce this production *only if* the current token, **CT**, is in Follow(**A**).

This makes sense since if we reduce α to **A**, we can't correctly match **CT** if **CT** can't follow **A**.

Shift/Reduce and Reduce/ Reduce Errors

If we have a parse state that contains the configurations

$\mathbf{A} \rightarrow \alpha \cdot$

$\mathbf{B} \rightarrow \beta \cdot \mathbf{a}\ \gamma$

and **a** in Follow(**A**) then there is an *unresolvable* shift/reduce conflict. This grammar can't be parsed.

Similarly, if we have a parse state that contains the configurations

$\mathbf{A} \rightarrow \alpha \cdot$

$\mathbf{B} \rightarrow \beta \cdot$

and Follow(**A**) $\cap$ Follow(**B**) $\neq \phi$, then the parser has an unresolvable reduce/reduce conflict. This grammar can't be parsed.