Reading Assignment

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

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

$$X \rightarrow A\ B \bullet C\ D$$

to represent the fact that we are trying to match the production $X \rightarrow A\ B \bullet C\ D$ 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:

$$X \rightarrow A\ B\ C\ D \bullet$$

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:

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

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

We may predict a λ-production:

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

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 → • A B C D

S → • e F g

S → • 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 → • 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 → • P Q R

A → • 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 → a •B d is in C &&
            B is a non-terminal)
        Add all configurations of
           the form   B → •g 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** $\rightarrow$ • **A b**) we first include all productions that rewrite A:

A $\rightarrow$ • **C D**

Now **C** productions are included:

C $\rightarrow$ • **D**

C $\rightarrow$ • **c**

Finally, the D production is added:

D $\rightarrow$ • **d**

The complete configuration set is:

S $\rightarrow$ • **A b**

A $\rightarrow$ • **C D**

C $\rightarrow$ • **D**

C $\rightarrow$ • **c**

D $\rightarrow$ • **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

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

$\mathbf{B} \rightarrow \beta \bullet \mathbf{a}\, \gamma$

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

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

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

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

Building Parse States

All the manipulations needed to build and complete configuration sets suggest that parsing may be slow—configuration sets need to be updated after each token is matched.

Fortunately, all the configuration sets we ever will need can be computed and tabled *in advance*, when a tool like Java Cup builds a parser.

The idea is simple. We first compute an initial parse state, s_0, that corresponds to predicting productions that expand the start symbol. We then just compute successor states for each token that might be scanned. A complete set of states can be computed. For typical

programming language grammars, only a few hundred states are needed.

Here is the algorithm that builds a complete set of parse states for a grammar:

```
StateSet BuildStates(){
 Let s0=Closure({S → •α, S → •β, ...});
 C={s0};
 while (not all states in C are marked){
  Choose any unmarked state, s, in C
  Mark s;
  For each X in
      terminals U nonterminals {
    if (GoTo(s,X) is not in C)
       Add GoTo(s,X) to C;
  }
 }
 return C;
}
```

Configuration Sets for CSX-Lite

State	Cofiguration Set
s_0	**Prog → •{ Stmts } Eof**
s_1	**Prog → { • Stmts } Eof** **Stmts → •Stmt Stmts** **Stmts → λ •** **Stmt → • id = Expr ;** **Stmt → • if (Expr) Stmt**
s_2	**Prog → { Stmts •} Eof**
s_3	**Stmts → Stmt • Stmts** **Stmts → •Stmt Stmts** **Stmts → λ •** **Stmt → • id = Expr ;** **Stmt → • if (Expr) Stmt**
s_4	**Stmt → id • = Expr ;**
s_5	**Stmt → if • (Expr) Stmt**

State	Cofiguration Set
s_6	**Prog → { Stmts } •Eof**
s_7	**Stmts → Stmt Stmts •**
s_8	**Stmt → id = • Expr ;** **Expr → • Expr + id** **Expr → • Expr - id** **Expr → • id**
s_9	**Stmt → if (• Expr) Stmt** **Expr → • Expr + id** **Expr → • Expr - id** **Expr → • id**
s_{10}	**Prog → { Stmts } Eof •**
s_{11}	**Stmt → id = Expr • ;** **Expr → Expr • + id** **Expr → Expr • - id**
s_{12}	**Expr → id •**
s_{13}	**Stmt → if (Expr •) Stmt** **Expr → Expr • + id** **Expr → Expr • - id**

State	Cofiguration Set
s_{14}	**Stmt → id = Expr ; •**
s_{15}	**Expr → Expr + • id**
s_{16}	**Expr → Expr - • id**
s_{17}	**Stmt → if (Expr) • Stmt** **Stmt → • id = Expr ;** **Stmt → • if (Expr) Stmt**
s_{18}	**Expr → Expr + id •**
s_{19}	**Expr → Expr - id •**
s_{20}	**Stmt → if (Expr) Stmt •**

Parser Action Table

We will table possible parser actions based on the current state (configuration set) and token.

Given configuration set C and input token T four actions are possible:

- Reduce i: The i-th production has been matched.
- Shift: Match the current token.
- Accept: Parse is correct and complete.
- Error: A syntax error has been discovered.

We will let A[C][T] represent the possible parser actions given configuration set C and input token T.

A[C][T] =
{Reduce i | i-th production is $\mathbf{A} \rightarrow \alpha$
and $\mathbf{A} \rightarrow \alpha \bullet$ is in C
and T in Follow(A) }
U (If ($\mathbf{B} \rightarrow \beta \bullet \mathbf{T}\ \gamma$ is in C)
{Shift} else ϕ)

This rule simply collects all the actions that a parser might do given C and T.

But we want parser actions to be unique so we require that the parser action always be *unique* for any C and T.

If the parser action isn't unique, then we have a shift/reduce error or reduce/reduce error. The grammar is then rejected as unparsable.

If parser actions are always unique then we will consider a shift of EOF to be an accept action.

An empty (or undefined) action for C and T will signify that token T is illegal given configuration set C.
A syntax error will be signaled.

LALR Parser Driver

Given the GoTo and parser action tables, a Shift/Reduce (LALR) parser is fairly simple:

```
void LALRDriver(){
  Push(S0);
  while(true){
  //Let S = Top state on parse stack
  //Let CT = current token to match
    switch (A[S][CT]) {
     case error:
        SyntaxError(CT);return;
     case accept:
        return;
     case shift:
        push(GoTo[S][CT]);
        CT= Scanner();
        break;
     case reduce i:
        //Let prod i = A→Y1...Ym
        pop m states;
        //Let S' = new top state
        push(GoTo[S'][A]);
        break;
} } }
```

Action Table for CSX-Lite

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
{	S																				
}		R3	S	R3				R2							R4						R5
if		S		S											R4			S			R5
(						S															
)													R8	S					R6	R7	
id		S		S					S	S					R4	S	S	S			
=					S																
+												S	R8	S					R6	R7	
-												S	R8	S					R6	R7	
;												S	R8						R6	R7	R5
eof							A														

GoTo Table for CSX-Lite

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
{	1																				
}			6																		
if		5		5														5			
(						9															
)														17							
id		4		4					12	12						18	19	4			
=					8																
+												15		15							
-												16		16							
;												14									
eof							10														
stmts		2		7																	
stmt		3		3																	
expr									11	13											

Example of LALR(1) Parsing

We'll again parse

`{ a = b + c; } Eof`

We start by pushing state 0 on the parse stack.

Parse Stack	Top State	Action	Remaining Input
0	Prog → •{ Stmts } Eof	Shift	`{ a = b + c; } Eof`
1 0	Prog → { • Stmts } Eof Stmts → • Stmt Stmts Stmts → λ • Stmt → • id = Expr ; Stmt → • if (Expr)	Shift	`a = b + c; } Eof`
4 1 0	Stmt → id • = Expr ;		`= b + c; } Eof`
8 4 1 0	Stmt → id = • Expr ; Expr → • Expr + id Expr → • Expr - id Expr → • id	Shift	`b + c; } Eof`

Parse Stack	Top State	Action	Remaining Input
12 8 4 1 0	Expr → id •	Reduce 8	+ c; } Eof
11 8 4 1 0	Stmt → id = Expr • ; Expr → Expr • + id Expr → Expr • - id	Shift	+ c; } Eof
15 11 8 4 1 0	Expr → Expr + • id	Shift	c; } Eof

Parse Stack	Top State	Action	Remaining Input
18 15 11 8 4 1 0	Expr → Expr + id •	Reduce 6	; } Eof
11 8 4 1 0	Stmt → id = Expr • ; Expr → Expr • + id Expr → Expr • - id	Shift	; } Eof
14 11 8 4 1 0	Stmt → id = Expr ; •	Reduce 4	} Eof

Parse Stack	Top State	Action	Remaining Input
3 1 0	Stmts → Stmt • Stmts Stmts → •Stmt Stmts Stmts → λ • Stmt → • id = Expr ; Stmt → • if (Expr) Stmt	Reduce 3	} Eof
7 3 1 0	Stmts → Stmt Stmts •	Reduce 2	} Eof
2 1 0	Prog → { Stmts •} Eof	Shift	} Eof
6 2 1 0	Prog → { Stmts } •Eof	Accept	Eof

Error Detection in LALR Parsers

In bottom-up, LALR parsers syntax errors are discovered when a blank (error) entry is fetched from the parser action table.

Let's again trace how the following illegal CSX-lite program is parsed:

```
{ b + c = a; } Eof
```

Parse Stack	Top State	Action	Remaining Input
0	Prog → •{ Stmts } Eof	Shift	{ b + c = a; } Eof
1 0	Prog → { • Stmts } Eof Stmts → • Stmt Stmts Stmts → λ • Stmt → • id = Expr ; Stmt → • if (Expr)	Shift	b + c = a; } Eof
4 1 0	Stmt → id • = Expr ;	Error (blank)	+ c = a; } Eof