What happens when LR(k) constructions fail?

If table construction reveals an inadequate state, one of the following must hold:

The grammar is ambiguous.

If the language is not itself inherently ambiguous, then perhaps the grammar can be modified to generate the same language, but unambiguously.

This is a task for human intelligence, as it's provably undecidable (i.e., there is no mechanical process to decide) that a grammar is ambiguous.

A method that works well is to identify the inadequate states, and then work into and out of the state to generate a string that has more than one derivation. The conflicts (identified, for example, by YACC) are helpful in this process.

Underfueled table construction

1. Generally, SLR is more powerful than LR(0); LALR is more powerful than SLR; LR is the most powerful (canonical) bottom-up parsing method.

2. Canonical LR parsers must form their reductions on top-of-stack. For some grammars (an example follows), no bounded amount of lookahead (bounded at table construction time) suffices to disambiguate some state.

A good exercise is to attempt adding nested procedures into the ANSI C grammar. foo(,,,) { } becomes problematic: One can't tell whether foo is a procedure definition or invocation until the arbitrarily distant opening brace is seen.
A grammar that is not LR(k) for any k

\[ S \rightarrow A\ a \]
\[ | B\ b \]
\[ A \rightarrow A\ d \]
\[ | d \]
\[ B \rightarrow B\ d \]
\[ | d \]

In the above grammar, a reduction must occur for the first "d" in the input, but the lookahead necessary for deciding whether to reduce \( A \rightarrow d \) or \( B \rightarrow d \) could be arbitrarily large.

If the right-hand sides of the first rules for \( A \) and \( B \) were reversed, then the grammar is LR(1), but the stack grows arbitrarily large at parse time.

Often the grammar can be modified to become LR(k), since this problem usually pertains to how the language is structured by the grammar.
Semantic processing

Also, there are often language constraints that are difficult or unwieldy to enforce syntactically.

For example, the ANSI C grammar essentially has a set of rules:

- Declaration -> Qualifiers id
- Qualifiers -> Qualifiers Qualifier
- Qualifier -> int
- int
- float
- static
- extern
- i
- Qualifier

A grammar that accommodates type information would involve some context, and such grammars are difficult to design and expensive to process. Viable approaches to this problem involve some form of semantic processing, performed during or shortly after parsing:

- Attribute grammars specify equations whose resolution essentially performs type checking.
- Symbol tables are the most common solution. Type information is entered when identifiers are declared, so that expression types can be subsequently checked.

There is still the issue of whether type checking occurs in the same pass over the input as syntactic checking. Some languages forbid the kinds of “forward” declarations that would require extra passes for type checking.

The grammar could be transformed to enforce the kind and number of qualifiers that are allowed, but this would increase the size of the grammar.

Another example would be the evaluation of an expression. If we restricted the size of its terms, each expression could be syntactically evaluated by a huge grammar. Taken further, any programming language can be processed by a finite-state machine if the program size is bounded.

Ultimately, issues of taste and efficiently dictate how and where language issues are addressed.
Ordering from a Chinese menu

The rules for a “correctly” placed order are:

1. At most one item may be selected from any column.
2. Some columns may be skipped.
3. At least one item must be chosen.
4. The items can be arbitrarily ordered.

The assignment is to rewrite the grammar to enforce the rules. This is exactly what’s needed to enforce C’s rules for declarations.

Solution

While some factoring of this grammar is possible, this example illustrates the tradeoff between grammar size and specificity of the parse.
Symbol tables

The symbol table tracks symbols and their types, where type information could be any property of a symbol relevant to subsequent activity in the compiler.

Such information typically includes:
- the basic type of a variable (ptr, int, char, float, struct, etc.);
- structure layout, pointer specifics, array information;
- initialization values;
- scope information.

static char *a[5];
is an array of 5 pointers to characters.

I provide the following symbol table access functions:

IncrNestLevel(): increase the nest level by one.
DecrNestLevel(): decrease the nest level by one.
EnterSymbol(M,name): enters the string name as a symbol of type M at the current nest level.
RetrieveSymbol(name): returns a pointer to the currently active declaration of name. If name is not active, an error message is produced and the parse is aborted.
ExistsSymbol(name): operates like RetrieveSymbol(), but instead of aborting, a NULL pointer is returned if name isn’t active.

I provide extra credit for those who implement their own, hash-based symbol table manager.

Copyright © 1994 Ron K. Cytron. All rights reserved ±8 1± SIGPLAN '94 COMPILER CONSTRUCTION TUTORIAL

Symbol tables

Essential information
1. Names;
2. Scope information;
3. Type information;
4. Storage specifics.

Issues
1. Programs typically contain a mix of very long and very short names (i vs. WindowMaxAccelScreenMouse()).
2. Type checking and code generation do not require access to all scopes at all times. Typically, access is required only to the current scope and its outer scopes. Even then, programs use the current and outermost scopes most frequently.

There are two popular methods of establishing symbol tables:
1. Make a separate pass over the program to create the symbol table;
2. Build the symbol table as you parse.

Given that one typically creates an abstract syntax tree anyway, it seems wise to defer symbol table creation to a separate pass. On the other hand, restructuring the grammar to simplify symbol table creation is a good exercise, and it is necessary for a one-pass compiler.
Semantic processing at parse time

Recall how an LR parser uses a stack to apply reductions:

\[
\begin{array}{c}
1 & E + T & 11 \\
2 & E & 2 \\
\end{array}
\]

\[E + T \leftarrow E\]

Our parse stack previously consisted of a stack, where \( a \) is a grammar symbol and \( n \) is a parse state. We now augment the stack to contain semantic information:

\[
\begin{array}{c}
\downarrow & a & \downarrow \\
\downarrow & n & \downarrow \\
\downarrow & s & \downarrow \\
\end{array}
\]

We can use this semantic stack to synthesize information during the parse. In our example, when \( E + T \) is reduced, we could add together the values associated with \( E \) and \( T \), pushing the sum on the stack along with the \( E \) replacing the \( E + T \):

\[
\begin{array}{c}
1 & E + T & 11 \\
2 & 47 & 11 \\
\end{array}
\]

\[E + T \leftarrow E\]

In \textit{YACC}, the grammar file can specify a union of types for the semantic stack, so that information of any form can be synthesized during the parse.
Synthesized attributes

With YACC, a segment of C code can be associated with each production. Given a rule

\[ A \rightarrow a_1 a_2 \ldots a_k \]

the segment of C code can refer to the semantic stack values of symbols as follows:

<table>
<thead>
<tr>
<th>Rule Symbol</th>
<th>Semantic Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a_1)</td>
<td>$1</td>
</tr>
<tr>
<td>(a_2)</td>
<td>$2</td>
</tr>
<tr>
<td>(\ldots)</td>
<td>$k</td>
</tr>
<tr>
<td>(A)</td>
<td>$$</td>
</tr>
</tbody>
</table>

so that a typical rule looks like

\[ A \rightarrow a_1 a_2 \ldots a_k \]

\(\{\$$ = \$3 + f(\$2)\}\)

More generally, one can reference any value still on the stack. In our example, this include information associated with F and G. Such grammars are called L-attributed.

Evaluating infix expressions

\[ S \rightarrow E \$
  \{\text{printf("Answer is \%d\n", \$1);}\} \]
\[ E \rightarrow E + T
  \{\$$ = \$1 + \$3;\}
  \mid T
  \{\$$ = \$1;\} \]
\[ T \rightarrow T* F
  \{\$$ = \$1 * \$3;\}
  \mid F
  \{\$$ = \$1;\} \]
\[ F \rightarrow (E)
  \{\$$ = \$2;\}
  \mid \text{const}
  \{\$$ = \$1;\} \]

Notice how the unit productions \(A \rightarrow B\) lead to simple copying of values up the parse tree. While these rules participate in disambiguating the grammar, they are not always conducive to semantic processing.
Grammars and semantic processing

There are usually many unambiguous grammars that generate a given programming language. In planning for semantic processing, it is often convenient to rewrite the grammar so that reductions and stack activity are conducive to the required actions.

Consider the grammar:

$$\text{Num} \rightarrow x \text{D}$$
$$\quad | \text{D}$$
$$\text{D} \rightarrow \text{D} \text{d}$$
$$\quad | \text{d}$$

Interpretation: a string of digits represents a base-10 number, unless the string is preceded by an 'x', in which case the string represents a base-8 number.

<table>
<thead>
<tr>
<th>String</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 4 7</td>
<td>347</td>
</tr>
<tr>
<td>x 3 4 7</td>
<td>231</td>
</tr>
</tbody>
</table>

We could compute the number by passing the list of digits up the tree, forming the answer at Num. We would prefer to compute the number as we reduce the digits, but this grammar's parse trees have the base information in the wrong place.
Rewriting the grammar

```
Num → x OctD $  
   {printf("Answer: %d\n",$2)}  
   |  DecD $  
   {printf("Answer: %d\n",$1)}  
DecD → DecD d  
   {$$ = (10 * $1) + $2;}  
   |  d  
   {$$ = $1;}  
OctD → OctD d  
   {$$ = (8 * $1) + $2;}  
   |  d  
   {$$ = $1;}  
```

---

```
State  d  x  $  DecD  OctD
1      4   2   5   
2      6   7   10  9   
3      8   7   4   5   
4      4   3   6   10  9  5   
5      10  9   11  12  13  14 
6      6   7   4   5   
7      3   6   3   5   
8      3   6   3   5   
9      3   6   3   5   
10     5   6   5   7   
```

Note: grammar is LR(0)

---

Another change!

Suppose we want the base itself to be part of the input:

<table>
<thead>
<tr>
<th>String</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 4 7</td>
<td>347</td>
</tr>
<tr>
<td>x 8 3 4 7</td>
<td>231</td>
</tr>
<tr>
<td>x 9 3 4 7</td>
<td>286</td>
</tr>
</tbody>
</table>

One possibility is to use a global variable:

```
Num → x B D $  
   {printf("Answer: %d\n",$3);}  
   |  Skip D $  
   {printf("Answer: %d\n",$2);}  
B → d  
   {Base = $1;}  
Skip → λ  
   {Base = 10;}  
D → D d  
   {$$ = (Base * $1) + $2;}  
   |  d  
   {$$ = $1;}  
```

Note that the reduction B → d is necessary to set the global variable. In the LR parse, this is the first reduction, so the base will indeed be set when the first D → D d rule is applied. But global variables are not very clean, especially if constructs could be nested so that global variables get overwritten.
Arriving at a good grammar

Let's engineer the tree we would like to see, and then construct the appropriate grammar. In the tree shown to the right, it's possible to synthesize the base up the tree.

At each reduction, we could know how to compute the new value to pass up the tree.

```
Num → D $  
   {printf("Answer: %d\n",$1.value);}  
D → D d  
   {$$.value = ($1.base * $1.value) + $2;  
    $$$.base = $1.base;}  
   |  
   B  
   {$$.base = $1;}  
B → x d  
   {$$ = $2;}
   |  
   x  
   {$$ = 10;}  
```

Back to declarations

The running example of converting a string of digits to a number is actually an abstraction of processing variable declarations in C.

We would like to enter the variables in the symbol table, along with their types, as we parse the input. Rewriting the ANSI C grammar to accomplish this is a good exercise. Note that PASCAL has its type information at the end, and so a right recursive rule can similarly accommodate that form of syntax.
An attributed grammar allows semantic equations whose terms depend on synthesized and inherited attributes. A classical use of attribute grammar systems is for the synthesis and use of type information.

As the declarations are parsed, a “symbol table” is synthesized up the parse tree. While processing the code of a procedure, this symbol and those from outer scopes are available as an inherited attribute.

While attribute grammars offer a clean mechanism for expressing semantics, such systems are usually slower than those involving only synthesized attributes, and one must still get the equations “right”. The Cornell Program Synthesizer is a popular and robust system for developing compilers based on attribute grammars [41, 31, 32].

The AST eliminates the scaffolding introduced to render the grammar unambiguous. Items such as temporary variables can be introduced into the AST to simplify subsequent activity (optimization, code generation).
Creating an AST

We can easily add actions to the grammar to create AST nodes and properly link these nodes to form the AST.

```
S → E$
E → E + T
  {$$ = MakeBinTree(PLUS, $1, $3);}
  | T
  {$$ = $1;}
T → T * F
  {$$ = MakeBinTree(TIMES, $1, $3);}
  | F
  {$$ = $1;}
F → ( E )
  {$$ = $2;}
  | const
  {$$ = MakeConst($1);}
  | id
  {$$ = MakeSymb($1);}
```

Free of clutter, the resulting tree can then be traversed to instantiate symbol tables, perform type checking, optimize the program, and generate code.

AST routines

typedef struct _TreeNode {
  struct {
    int linenumber;
    int colnumber;
  } sourceinfo;
  NodeInfo info;
  struct _TreeNode *child;
  struct _TreeNode *sibling;
  struct _TreeNode *head;
  struct _TreeNode *parent;
  struct _TreeNode *leftsib;
} TreeNode;

NodeInfo is a union of tree node information: symbol table pointers, integer values, operator types, etc.

MakeFamily(parent, sibs): adopts sibs into the parent’s family, returning the parent.

MakeSiblings(c1, c2): units siblings c1 and c2, returning the end of the resulting list (shown below).

MakeOperatorNode(opnum): creates an operator node, where opnum is the “name” of a “token”.

MakeIntegerNode(intval): creates an integer node with value intval.

MakeStringNode(str): creates a string node with value str.

MakeSymbolNode(sym): creates a symbol reference node to sym.
The AST is shown to the right, with indentation reflecting tree depth. Note the regular structure:
- functions and inline procedures are represented similarly.
- an if-then structure is represented as an if-then-else with trivial "else" code.

The above list is created by the actions shown to the left. The first number in the list is the base, and the subsequent numbers are the digits as parsed from left to right.
Left and right values of identifiers

Named for their interpretation with respect to "=", the left value of an identifier is its location while the right value is the contents of the identifier.

**L vs. R values**

The positioning of identifiers with respect to various operators in C indicates which value is desired:

- `X=` The storage location of X
- `EY` The value stored at Y
- `* Z` The value at Z, treated as a storage location
- `& W` The address of W, treated as a value

**Type checking**

The actual meaning of the identifier is dependent on its context.

The meaning of `+` in the above program depends on the types of Y, Z, and X. In languages that allow operator overloading, even the meaning of `+` becomes suspect.

Notice the dual role of the operator `*` (in the C language), which is nicely disambiguated using the proper grammar.
C left and right values

<table>
<thead>
<tr>
<th>Form</th>
<th>Expects</th>
<th>Produces</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=b</td>
<td>LV(a), RV(b)</td>
<td>RV</td>
</tr>
<tr>
<td>* c</td>
<td>RV(c)</td>
<td>LV</td>
</tr>
<tr>
<td>&amp;d</td>
<td></td>
<td>RV</td>
</tr>
</tbody>
</table>

This grammar produces structures where the interpretation of left and right values is clear.

Moreover, the rule R → L is applied when a left value “becomes” a right value through dereferencing.

The grammar correctly precludes strings like “3=x” and “&z=y”.

Table construction for this grammar fails for SLR because “=” can follow an R.

But there’s no sentential form that begins “R=”; the R must be preceded by an * as in “*R=”. The LR(1) construction can create a suitable parse table. The grammar is also LALR(1) (YACC can handle this grammar).

Examples

```
x=y
```

- Push x
- Push y
- Fetch
- Store

```
S
/
L = R
|
S
/
L
/
R
|
S
/
L
/
R
/
R
/
R
/
R
/
R
/
R
```

The * and & have no effect on code generation: they merely change the type of an expression. The language design is biased towards the most prevalent “x=y” form.
More on left and right values

Unfortunately, the syntactic rules for C do not allow a grammar-based approach to left and right value disambiguation. However, the rules related to =, &, and * can be applied just as easily to the parse tree, using attribute grammars or an additional (bottom-up) pass over the tree.

SetV(node, kind): asserts the left or right valuedness of node.
ExpectLV(node): expects that node is a left value.
ExpectRV(node): expects that node is a right value.
Convert(node, how): attempts to convert node into a left or right value.

As our grammar indicates, the only conversion that makes sense is a left to right value conversion, which is basically a dereference. In a call-by-reference language, however, a right value could become a left value by introducing a temporary. Show below are the parse trees before and after the extra bottom-up pass.
Data types and compatible operations

The type checking phase of a compiler is traditionally responsible for establishing the semantic well-formedness of operations and data. Where language standards allow flexibility (some would say sloppiness) with respect to type consistency, compilers are charged with introducing implicit or explicit conversion operations to allow operations on otherwise unsuitable data.

Most languages offer a host of basic types, which are usually (though not always) supported by target instruction sets:

- integers;
- floating point;
- Boolean-valued \{ true, false \};
- character.

Most languages also allow the introduction of new types based on old ones:

- tuples (records, structs);
- maps (functions, arrays);
- sets.

Proponents of strongly-typed languages, where data and operations must adhere rigidly to type consistency, claim that when soundly checked at compile-time, their programs are less likely to contain bugs.

Type checking

As with left and right value determination, type checking can be performed as a bottom-up pass over the parse (or abstract syntax) tree.

A straightforward (i.e., highly localized) scheme operates as follows. At the tree’s leaves are found the atomic elements such as constants, identifiers, and function calls. Each of these asserts its type, based on syntactic (“x” vs. ‘x’) or contextual (declared) information.

At each internal node \( X \)

1. the subtrees of \( X \) are checked for type compatibility: this depends on the operation contained in \( X \);
2. conversion operations are inserted as necessary;
3. the type of \( X \) is determined.
The C language has cast operations, that assert the type of an expression. But conversions can also occur in uncast expressions, which can lead to confusion.

Are the above casts necessary? It depends on whether we regard type equivalence as a structural property or as a property of the name used in the declaration. In the above examples, $x$, $y$, and $foo$ are all structurally represented as an integer.

The use of $+$ on $foo$ could also be problematic, if an enum data type cannot be the target of $+$.

Pointers are an interesting example, since they are all structurally the same. Good language design and programming practice suggest distinguishing between pointers to different types.

C castigates those who fail to cast between pointers of different types.