CS415 Compilers

Syntax Analysis

Part 9

Context-Sensitive Analysis

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University
Announcements

• Fifth homework has been posted
• Second project has been posted
Bottom-up Parsing
(Syntax Analysis)

EAC Chapters 3.4
ALSU Chapter 4.5
LR(0) versus SLR(1) versus LR(1)

Example:

\[ S' \rightarrow S \]
\[ S \rightarrow S ; a \mid a \]

LR(0) ? s0 = \{[S' \rightarrow S.], [S \rightarrow . S ; a], [S \rightarrow . a]\}
\[ s1 = \text{goto}(s0,S) = \{[S'\rightarrow S.], [S \rightarrow S . ; a]\} \text{ **conflict**} \]

LR(1) ? YES - check at home or in recitation

SLR(1) ? SIMPLE LR(1)  FOLLOW (S) = \{eof, ; \}
\[ s1 = \{[S' \rightarrow S . , eof], [S \rightarrow S . ; a, \{eof, ;\}]\} \text{ **no conflict**} \]

SLR(1): add FOLLOW(A) to each LR(0) item [A \rightarrow \gamma^*] as its second component: [A \rightarrow \gamma^* , a], \forall a \in \text{FOLLOW}(A);

Note: Can also add to other items, but does not really matter.
LR(0) versus SLR(1) versus LR(1)

1: \( S' \rightarrow S \)
2: \( S \rightarrow S ; a \)
3: \( S \rightarrow a \)

**LR(0):**

\[ S_0 = \{[S' \rightarrow S], [S \rightarrow S ; a], [S \rightarrow a]\} \]
\[ S_1 = \text{Goto}(S_0, S) = \{[S' \rightarrow S], [S \rightarrow S ; a]\} \]
\[ S_2 = \text{Goto}(S_0, a) = \{[S \rightarrow a]\} \]
\[ S_3 = \text{Goto}(S_1, ;) = \{[S \rightarrow S ; a]\} \]
\[ S_4 = \text{Goto}(S_3, a) = \{[S \rightarrow S ; a]\} \]

**SLR(1):**

Follow\((S') = \{\text{eof}\} \)
Follow\((S) = \{\text{eof, ;}\} \)

Grammar is SLR(1)

**LR(0) parse table**

<table>
<thead>
<tr>
<th></th>
<th>S0</th>
<th>S1</th>
<th>S2</th>
<th>S3</th>
<th>S4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>shift</td>
<td>**shift/reduce **</td>
<td><strong>reduce rule 3</strong></td>
<td>shift</td>
<td><strong>reduce rule 2</strong></td>
</tr>
</tbody>
</table>

**Grammar is not LR(0)!**

**SLR(1) parse table**

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>;</th>
<th>eof</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>shift</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S1</td>
<td>shift</td>
<td>reduce rule 1 (accept)</td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td>reduce rule 3</td>
<td>reduce rule 3</td>
<td></td>
</tr>
<tr>
<td>S3</td>
<td>shift</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S4</td>
<td>reduce rule 2</td>
<td>reduce rule 2</td>
<td></td>
</tr>
</tbody>
</table>
LALR(1) versus LR(1)

Example:

\[ S' \rightarrow S \]
\[ S \rightarrow aAd \mid bBd \mid aBe \mid bAe \]
\[ A \rightarrow c \]
\[ B \rightarrow c \]

LR(0)?

LR(1)?

LALR(1)?

**LALR(1):** Merge two sets of LR(1) items (states), if they have the same **core**.

**core** of set of LR(1) items: set of LR(0) items derived by ignoring the lookahead symbols
LALR(1) versus LR(1)

\[ s_0 = \text{Closure}\left(\{S' \rightarrow .S, \text{eof}\}\right) = \{[S \rightarrow .aAd, \text{eof}], [S \rightarrow .aBe, \text{eof}], [S \rightarrow .bAe, \text{eof}], [S \rightarrow .bBd, \text{eof}], [S' \rightarrow .S, \text{eof}]\} \]

\[ s_1 = \text{Closure}(\text{GoTo}\ (s_0, a)) = \{[S \rightarrow a . Ad, \text{eof}], [S \rightarrow a . Be, \text{eof}], [A \rightarrow .c, d], [B \rightarrow .c, e]\} \]

\[ s_2 = \text{Closure}(\text{GoTo}\ (s_0, b)) = \{[S \rightarrow b . Ae, \text{eof}], [S \rightarrow b . Bd, \text{eof}], [A \rightarrow .c, e], [B \rightarrow .c, d]\} \]

\[ s_3 = \text{Closure}(\text{GoTo}\ (s_1, c)) = \{[A \rightarrow c . , d], [B \rightarrow c . , e]\} \]

\[ s_4 = \text{Closure}(\text{GoTo}\ (s_2, c)) = \{[A \rightarrow c . , e], [B \rightarrow c . , d]\} \]

There are other states that are not listed here!

Grammar is LR(1), but not LALR(1), since collapsing \( s_3 \) and \( s_4 \) (same core) will introduce reduce-reduce conflict.
LALR(1) versus LR(1)

Example:

LR(0) ? NO

LR(1) ? YES

LALR(1) ? NO, since introduces a reduce/reduce conflict

**LALR(1):** Merge two sets of LR(1) items (states), if they have the same **core**.

**core** of set of LR(1) items: set of LR(0) items derived by ignoring the lookahead symbols

**FACT:** collapsing LR(1) states into LALR(1) states cannot introduce shift/reduce conflicts
Three options:

• **Combine terminals such as number & identifier, + & -, *, /**
  -> Directly removes a column, may remove a row
  -> For expression grammar, 198 (vs. 384) table entries

• **Combine rows or columns** *(table compression)*
  -> Implement identical rows once & remap states
  -> Requires extra indirection on each lookup
  -> Use separate mapping for ACTION & for GOTO

• **Use another construction algorithm**
  -> Both LALR(1) and SLR(1) produce smaller tables
  -> Implementations are readily available
Finding Reductions

LR$(k)$ ⇒ Each reduction in the parse is detectable with
- the complete left context,
- the reducible phrase, itself, and
- the $k$ terminal symbols to its right

LL$(k)$ ⇒ Parser must select the next rule based on
- The complete left context
- The next $k$ terminals

Thus, LR$(k)$ examines more context
## Summary

### Advantages

<table>
<thead>
<tr>
<th>Top-down recursive descent</th>
<th>LR(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Easy to implement</td>
<td>Fast</td>
</tr>
<tr>
<td>Good locality (fast)</td>
<td>Deterministic langs.</td>
</tr>
<tr>
<td>Simplicity</td>
<td>Automatable (tool support)</td>
</tr>
<tr>
<td>Easy to embed actions</td>
<td>Left associativity</td>
</tr>
<tr>
<td>(code access)</td>
<td></td>
</tr>
</tbody>
</table>

### Disadvantages

- Hand-coded
- High maintenance
- Right associativity
- Large working sets
- Large table sizes
Hierarchy of Context-Free Languages

Context-free languages

Deterministic languages (LR(\(k\)))

LL(\(k\)) languages

LL(1) languages

Simple precedence languages

Operator precedence languages

\(LR(k) \equiv LR(1)\)

The inclusion hierarchy for context-free languages
The inclusion hierarchy for context-free grammars

- Operator precedence includes some ambiguous grammars
- LL(1) is a subset of SLR(1)
YACC: parse.y (preview project #2)

parse.y:

```c
{%
#include <stdio.h>
#include "attr.h"
int yylex();
void yyerror(char * s);
#include "symtab.h"
%

%union {tokentype token; }

%token PROG PERIOD PROC VAR ARRAY RANGE OF
%token INT REAL DOUBLE WRITELN THEN ELSE IF
%token BEG END ASG NOT
%token EQ NEQ LT LEQ GEQ GT OR EXOR AND DIV NOT
%token ID CCONST ICONST RCONST

%start program

%%
program : PROG ID ';' block PERIOD ;
block : BEG ID ASG ICONST END ;
%
void yyerror(char* s) {
    fprintf(stderr,"%s\n",s);
}
int main() {
    printf("1\t");
    yyparse();
    return 1;
}

Will be included verbatim in parse.tab.c

List of tokens

CFG rules

Main program and “helper” functions; may contain initialization code of global structures. Will be included verbatim in parse.tab.c

Will be included verbatim in parse.tab.c

```
The problem: parser encounters an invalid token
Goal: Want to parse the rest of the file

Basic idea (panic mode):
→ Assume something went wrong while trying to find handle for nonterminal A
→ Pretend handle for A has been found; pop “handle”, skip over input to find terminal that can follow A

Restarting the parser (panic mode):
→ find a restartable state on the stack (has transition for nonterminal A)
→ move to a consistent place in the input (token that can follow A)
→ perform (error) reduction (for nonterminal A)
→ print an informative message
Error Recovery in YACC

Yacc’s (bison’s) error mechanism (note: version dependent!)
• designated token error
• used in error productions of the form
  \[ A \rightarrow \text{error} \ \alpha \] // basic case
• \( \alpha \) specifies synchronization points

When error is discovered
• pops stack until it finds state where it can shift the error token
• resumes parsing to match \( \alpha \)
  special cases:
  \( \rightarrow \alpha = w \), where \( w \) is string of terminals: skip input until \( w \) has been read
  \( \rightarrow \alpha = \varepsilon \): skip input until state transition on input token is defined
• error productions can have actions
cmpdstmt: BEG stmt_list END  
stmt_list : stmt  
| stmt_list ';' stmt  
| error { yyerror("n***Error: illegal statement
");}  

This should  
• throw out the erroneous statement  
• synchronize at “;” or “end” (implicit: \( \alpha = \varepsilon \))  
• writes message “***Error: illegal statement” to stderr  

Example: begin a & 5 | hello ; a := 3 end  
         \[\uparrow \quad \uparrow\] resume parsing  
         ***Error: illegal statement
Context-Sensitive Analysis

EaC Chapter 4
ALSU Chapter 5
There is a level of correctness that is deeper than grammar

```c
fie(a,b,c,d)
    int a, b, c, d;
{ ... }

fee() {
    int f[3],g[1],
    h, i, j, k;
char *p;
    fie(h,i,"ab",j, k);
    k = f * i + j;
    h = g[17];
    printf("<%s,%s>.\n",
          p,q);
    p = 10;
}
```

**What is wrong with this program?**
*(let me count the ways …)*
There is a level of correctness that is deeper than grammar.

```c
fie(a,b,c,d)
    int a, b, c, d;
{ ... }

fee() {
    int f[3], g[1],
        h, i, j, k;
    char *p;
    fie(h, i, "ab", j, k);
    k = f * i + j;
    h = g[17];
    printf("%s,%s.\n", p, q);
    p = 10;
}
```

What is wrong with this program? *(let me count the ways …)*

- declared g[1], used g[17]
- wrong number of args to fie()
- “ab” is not an int
- wrong dimension on use of f
- undeclared variable q
- 10 is not a character string

All of these are
“deeper than syntax”

To generate code, we need to understand its meaning!
These questions are part of context-sensitive analysis

- Answers depend on “values”, i.e., something that needs computation; not parts of speech
- Questions & answers involve non-local information

How can we answer these questions?

- Use formal methods
  - Context-sensitive grammars
  - Attribute grammars
    (attributed grammars)
- Use ad-hoc techniques
  - Symbol tables
  - Ad-hoc code
    (action routines)

In scanning & parsing, formalism won; somewhat different story here.
Telling the story

- The attribute grammar formalism is important
  - Succinctly makes many points clear
  - Sets the stage for actual, *ad-hoc* practice (e.g.: yacc/bison)

- The problems with attribute grammars motivate practice
  - Non-local computation
  - Need for centralized information

We will cover attribute grammars, then move on to *ad-hoc* ideas (syntax-directed translation schemes)
Attribute grammars

Syntax-directed translation

Type checking

Symbol tables

Read EaC: Chapters 5.1 - 5.3