Lesson 11

- Syntax-Directed Translation (cont’d)
- Parser generators: Yacc/Bison
Summary

• Syntax-Directed Definitions (Attribute Grammars)
  – Enrich CFG’s with Attributes and Semantic Rules
  – S-attributed SDD’s: synthesized attributes only
    • Attributes computed with postorder traversal
  – L-attributed SDD’s: also (constrained) inherited attributes
    • Attributes computed with left-to-right, depth-first traversal

• Syntax-Directed Translation Schemes
  – Embed semantic actions of SDD’s in productions
  – Sometimes allow to compute the translation without building the whole parsing tree

• Implementation of S-attributed SDD’s for LR grammars with bottom-up (LR) parsing: semantic actions are placed at the end of the corresponding production
Using Translation Schemes for L-Attributed Definitions

- An L-attributed SDD for a grammar that can be parsed top-down (LL) can be implemented using Translation Schemes

1. Embed actions that compute inherited attributes for nonterminal A immediately before A

2. Place actions that compute a synthesized attribute for the head of a production at the end of the body of that production
Using Translation Schemes for L-Attributed Definitions

Production | Semantic Rule
---|---
\( D \rightarrow TL \) | \( L.\text{in} := T.\text{type} \)
\( T \rightarrow \text{int} \) | \( T.\text{type} := \text{‘integer’} \)
\( T \rightarrow \text{real} \) | \( T.\text{type} := \text{‘real’} \)
\( L \rightarrow L_1, \text{id} \) | \( L_1.\text{in} := L.\text{in}; \text{addtype}(\text{id}.\text{entry}, L.\text{in}) \)
\( L \rightarrow \text{id} \) | \( \text{addtype}(\text{id}.\text{entry}, L.\text{in}) \)

Translation Scheme

\( D \rightarrow T \{ L.\text{in} := T.\text{type} \} L \)
\( T \rightarrow \text{int} \{ T.\text{type} := \text{‘integer’} \} \)
\( T \rightarrow \text{real} \{ T.\text{type} := \text{‘real’} \} \)
\( L \rightarrow \{ L_1.\text{in} := L.\text{in} \} L_1, \text{id} \{ \text{addtype}(\text{id}.\text{entry}, L.\text{in}) \} \)
\( L \rightarrow \text{id} \{ \text{addtype}(\text{id}.\text{entry}, L.\text{in}) \} \)
Recursive-Descent Parsing (Recap)

• Grammar must be LL(1)
• Every nonterminal has one (recursive) procedure
• When a nonterminal has multiple productions, the input look-ahead is used to choose one
• Note: the procedures have no parameters and no result

\[ expr \rightarrow term \, rest \]
\[ rest \rightarrow + \, term \, rest \]
\[ \mid - \, term \, rest \]
\[ \mid \varepsilon \]
\[ term \rightarrow id \]

\[ procedure \ rest(); \]
\[ \quad begin \]
\[ \quad \quad if \ lookahead \ in \ FIRST(+ \, term \, rest) \ then \]
\[ \quad \quad \quad match('+'); \ term(); \ rest() \]
\[ \quad \quad else \ if \ lookahead \ in \ FIRST(- \, term \, rest) \ then \]
\[ \quad \quad \quad match('-'); \ term(); \ rest() \]
\[ \quad \quad else \ if \ lookahead \ in \ FOLLOW(rest) \ then \]
\[ \quad \quad \quad return \]
\[ \quad \quad \quad else \ error() \]
\[ \quad end; \]
Implementing L-Attributed Definitions in Recursive-Descent Parsers

- Attributes are passed as arguments to procedures (inherited) or returned (synthesized)
- Procedures store computed attributes in local variables

\[
D \rightarrow T \{ L.in := T.type \} L \\
T \rightarrow \text{int} \{ T.type := \text{‘integer’} \} \\
T \rightarrow \text{real} \{ T.type := \text{‘real’} \}
\]

```c
void D()
{
    Type Ttype = T();
    Type Lin = Ttype;
    L(Lin);
}

Type T()
{
    Type Ttype;
    if (lookahead == INT)
    {
        Ttype = TYPE_INT;
        match(INT);
    }
    else if (lookahead == REAL)
    {
        Ttype = TYPE_REAL;
        match(REAL);
    }
    else error();
    return Ttype;
}

void L(Type Lin)
{
    ... 
}
```

- Input:
  - inherited attribute
- Output:
  - synthesized attribute
Implementing L-Attributed Definitions in Top-Down Table-Driven Parsers

• The stack will contain, besides grammar symbols, *action-records* and *synthesize-records*

• Inherited attributes of $A$ are placed in $A$’s record
  – The code computing them is in a record above $A$

• Synthesized attributes of $A$ are placed in a record just below $A$

• It may be necessary to make copies of attributes to avoid that they are popped when still needed
Implementing L-Attributed Definitions for LL grammars in Bottom-Up Parsers

- Remove any embedded action with marking
  
  nonterminal: \( A \rightarrow \alpha \{ \text{act} \} \beta \) becomes
  
  \( A \rightarrow \alpha N \beta \)
  
  \( N \rightarrow \varepsilon \{ \text{act}' \} \)

  where act’:

  - Copies as inherited attributes of \( N \) any attribute of \( A \), \( \alpha \) needed by act

  - Computes attributes like act, making them synthesized for \( N \)

- Fact: if the start grammar was LL, the new one is LR

- Note: act’ accesses attributes out of its production!
  This works, as they are (deeper) in the LR stack
Parser Generators: ANTLR, Yacc, and Bison

• **ANTLR tool**
  – Generates LL($k$) parsers

• **Yacc (Yet Another Compiler Compiler)**
  – Generates LALR parsers

• **Bison**
  – Improved version of Yacc (GNU project)
Creating an LALR(1) Parser with Yacc/Bison

yacc specification `spec.y` -> Yacc or Bison compiler -> `y.tab.c`

`y.tab.c` -> C compiler -> `a.out`

`a.out` -> output stream

input stream -> `a.out`
Yacc Specification

- A yacc specification consists of three parts:

  - yacc declarations, and C declarations within %{  %}
  %
  translation rules  (productions + semantic actions)
  %
  user-defined auxiliary procedures

- The translation rules are productions with actions:

  production_1  { semantic action_1 }
  production_2  { semantic action_2 }
  ...
  production_n  { semantic action_n }
Writing a Grammar in Yacc

• Production  $head \rightarrow body_1 \mid body_2 \mid \ldots \mid body_n \mid \varepsilon$
  becomes in Yacc

  \[
  head : \quad body_1 \quad \{ \text{semantic action}_1 \} \\
       | \quad body_2 \quad \{ \text{semantic action}_2 \} \\
       \ldots \\
       | \quad /* \text{empty} */ \\
  ;
  \]

• Tokens (terminals) can be:
  – Quoted single characters, e.g. ‘+’, with corresponding ASCII code
  – Identifiers declared as tokens in the declaration part using
    \%token TokenName

• Nonterminals:
  – Arbitrary strings of letters and digits (not declared as tokens)
Semantic Actions and Synthesized Attributes

- **Semantic actions** are sequences of C statements, and may refer to values of the *synthesized attributes* of terminals and nonterminals in a production:

  \[ X : Y_1 \ Y_2 \ Y_3 \ ... \ Y_n \{ \text{action} \} \]

  - `$$` refers to the value of the attribute of \( X \)
  - `$i` refers to the value of the attribute of \( Y_i \)

- For example

  ```
  factor : '(' expr ')' \{ $$=$2; \}
  ```

  ```
  factor.val=x
  ( expr.val=x )
  $$=$2
  ```

- The values associated with tokens (terminals) are those returned by the lexer.
An S-attributed Grammar for a simple desk calculator

The grammar
line → expr ‘\n’
expr → expr + term | term
term → term * factor | factor
factor → (expr) | DIGIT

Also results in definition of
#define DIGIT xxx

%token DIGIT
%%
line : expr ‘\n’ { printf("= %d\n", $1); }

expr : expr ‘+’ term { $$ = $1 + $3; }
    | term { $$ = $1; }

term : term ‘*’ factor { $$ = $1 * $3; }
    | factor { $$ = $1; }

factor : ‘(’ expr ‘)’ { $$ = $2; }
    | DIGIT { $$ = $1 }
A simple desk calculator

```c
#include <ctype.h>

%token DIGIT

line  : expr '\n'  { printf("= %d\n", $1); }

expr  : expr '+' term   { $$ = $1 + $3; }
      | term         { $$ = $1; }

term  : term '*' factor  { $$ = $1 * $3; }
      | factor       { $$ = $1; }

factor : '(' expr ')'   { $$ = $2; }
       | DIGIT        { $$ = $1; }

int yylex()
{ int c = getchar();
  if (isdigit(c))
    { yylval = c-'0';
      return DIGIT;
    }
  return c;
}
```

Attribute of token (stored in `yylval`)

Very simple lexical analyzer invoked by the parser

The grammar

- `line → expr \n`
- `expr → expr + term | term`
- `term → term * factor | factor`
- `factor → (expr) | DIGIT`
# Bottom-up Evaluation of S-Attributed Definitions in Yacc

<table>
<thead>
<tr>
<th>Stack</th>
<th>val</th>
<th>Input</th>
<th>Action</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>_</td>
<td>$3*5+4n$</td>
<td>shift</td>
<td>($$ = $1)</td>
</tr>
<tr>
<td>$3</td>
<td>3</td>
<td>*5+4n$</td>
<td>reduce (F \rightarrow \text{digit})</td>
<td>($$ = $1)</td>
</tr>
<tr>
<td>$F</td>
<td>3</td>
<td>*5+4n$</td>
<td>reduce (T \rightarrow F)</td>
<td>($$ = $1)</td>
</tr>
<tr>
<td>$T</td>
<td>3</td>
<td>*5+4n$</td>
<td>shift</td>
<td>($$ = $1)</td>
</tr>
<tr>
<td>$T*$</td>
<td>3 _</td>
<td>5+4n$</td>
<td>shift</td>
<td>($$ = $1)</td>
</tr>
<tr>
<td>$T*5$</td>
<td>3 _ 5</td>
<td>+4n$</td>
<td>reduce (F \rightarrow \text{digit})</td>
<td>($$ = $1)</td>
</tr>
<tr>
<td>$T*F$</td>
<td>3 _ 5</td>
<td>+4n$</td>
<td>reduce (T \rightarrow T*F)</td>
<td>($$ = $1 * $3)</td>
</tr>
<tr>
<td>$T$</td>
<td>15</td>
<td>+4n$</td>
<td>reduce (E \rightarrow T)</td>
<td>($$ = $1)</td>
</tr>
<tr>
<td>$E$</td>
<td>15</td>
<td>+4n$</td>
<td>shift</td>
<td>($$ = $1)</td>
</tr>
<tr>
<td>$E+$</td>
<td>15 _</td>
<td>4n$</td>
<td>shift</td>
<td>($$ = $1)</td>
</tr>
<tr>
<td>$E+4$</td>
<td>15 _ 4</td>
<td>n$</td>
<td>reduce (F \rightarrow \text{digit})</td>
<td>($$ = $1)</td>
</tr>
<tr>
<td>$E+F$</td>
<td>15 _ 4</td>
<td>n$</td>
<td>reduce (T \rightarrow F)</td>
<td>($$ = $1)</td>
</tr>
<tr>
<td>$E+T$</td>
<td>15 _ 4</td>
<td>n$</td>
<td>reduce (E \rightarrow E+T)</td>
<td>($$ = $1 + $3)</td>
</tr>
<tr>
<td>$E$</td>
<td>19</td>
<td>n$</td>
<td>shift</td>
<td>($$ = $1)</td>
</tr>
<tr>
<td>$E+n$</td>
<td>19 _</td>
<td>$</td>
<td>reduce (L \rightarrow E\ n)</td>
<td>($$ = $1)</td>
</tr>
<tr>
<td>$L$</td>
<td>19</td>
<td>$</td>
<td>accept</td>
<td>($$ = $1)</td>
</tr>
</tbody>
</table>
Dealing With Ambiguous Grammars

• By defining operator precedence levels and left/right associativity of the operators, we can specify ambiguous grammars in Yacc, such as

\[
E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid (E) \mid -E \mid \text{num}
\]

• Yacc resolves conflicts, by default, as follows:
  
  – **Reduce/reduce** conflict: precedence to first production in the specification
  
  – **Shift/reduce** conflict: precedence to shift
    • ok for *if-then-else*
    • infix binary operators are handled as right-associative
%token NUMBER
%
lines : expr '\n'       { printf("= %g\n", $1); }
expr  : expr '+' expr              { $$ = $1 + $3; }
| expr '*' expr              { $$ = $1 * $3; }
| NUMBER

• bison’s warning:
  conflicts: 4 shift/reduce
Dealing With Ambiguous Grammars

• To define precedence levels and associativity in Yacc’s declaration part, list tokens in order of increasing precedence, prefixed by `%left` or `%right`:
  - `%left `+` ’-’ `//same precedence, associate left
  - `%left `*` `’/’
  - `%right UMINUS

• If tokens have precedence, productions also have, equal to that of the rightmost terminal in the body. In this case:
  - **Shift/reduce** conflict are resolved with **reduce** if the production has higher precedence than the input symbol, or if they are equal and are left-associative.
Example: PlusTimesCalculator

```plaintext
%token NUMBER    /* tokens listed in increasing order of precedence */
%left '+'
%left '*'
%%
lines : expr \n'     { printf("= %g\n", $1); }
expr  : expr '+' expr         { $$ = $1 + $3; }
       | expr '*' expr         { $$ = $1 * $3; }
       | NUMBER
   ;
%%
```

- No warnings by bison

```
> ./PlusTimesCalculator-flat
1+2*3+4*5
= 27 /* correct precedence */
```

```
2 expr: expr .'+' expr
2   | expr '+' expr .
3   | expr .'*' expr

'*' shift, and go to state 6

$default reduce using rule 2 (expr)
```
A more advanced calculator

{%
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double
%
%token NUMBER  /* tokens listed in increasing order of precedence */
%left ‘+’ ‘-’
%left ‘*’ ‘/’
%right UMINUS  /* fake token with highest precedence, used below */
%%
lines  : lines expr ‘\n’  { printf("= %.9g\n", $2); }
  | lines ‘\n’
  | /* empty */
  ;
expr   : expr ‘+’ expr       { $$ = $1 + $3; }
  | expr ‘-’ expr       { $$ = $1 - $3; }
  | expr ‘*’ expr       { $$ = $1 * $3; }
  | expr ‘/’ expr       { $$ = $1 / $3; }
  | ‘(’ expr ‘)’       { $$ = $2; }
  | ‘-’ expr %prec UMINUS { $$ = -$2; } /* rule with highest precedence */
  | NUMBER
  ;
%%
Double type for attributes and \texttt{yy\_val}

Double type for attributes and \texttt{yy\_val}
A more advanced calculator (cont’d)

```c
%%
int yylex()
{
  int c;
  while ((c = getchar()) == ' ')
  ;
  if ((c == '.') || isdigit(c))
  {
    ungetc(c, stdin);
    scanf("%lf", &yylval);
    return NUMBER;
  }
  return c;
}
int main()
{
  if (yyparse() != 0)
    fprintf(stderr, "Abnormal exit\n");
  return 0;
}
int yyerror(char *s)
{
  fprintf(stderr, "Error: %s\n", s);
}
```

Crude lexical analyzer for fp doubles and arithmetic operators

Run the parser

Invoked by parser to report parse errors
Dealing With Ambiguous Grammars (summary)

• Yacc does not report about conflicts that are solved using user-defined precedences
• It reports conflicts that are resolved with the default rules
• To visit the automaton and the LALR parsing table generated, execute Bison/Yacc with option –v, and read the `<filename>.output` file
• This allows to see where conflicts were generated, and if the parser resolved them correctly
• Graphical representation of the automaton using Bison/Yacc with option –g. Output should be in `dot` format
Combining Lex/Flex with Yacc/Bison

**Yacc or Bison compiler**

- **yacc specification**
  - yacc.y

- **Lex specification**
  - lex.l
  - y.tab.h

- **Lex or Flex compiler**
  - lex.yy.c

- **C compiler**
  - y.tab.h
  - lex.yy.c

- **a.out**
  - a.out

Input stream

Output stream
Lex Specification for Advanced Calculator

```c
%option noyywrap
%
#define YYSTYPE double
#include "y.tab.h"

extern double yylval;
%
number [0-9]+\.?|[0-9]*\.[0-9]+%

[ ] { /* skip blanks */ }
{number} { sscanf(yytext, "%lf", &yylval);
   return NUMBER;
   }

\n|. { return yytext[0]; } 
```

Generated by Yacc, contains
```c
#define NUMBER xxx
```

Defined in `y.tab.c`

```
yacc -d example2.y
lex example2.l
gcc y.tab.c lex.yy.c
./a.out
```

```
bison -d -y example2.y
flex example2.l
gcc y.tab.c lex.yy.c
./a.out
```
Error Recovery in Yacc

- Based on error productions of the form \( A \rightarrow \text{error } \alpha \)

```c
%{
...
}%
...
%
lines : lines expr \n \{ printf("%g\n", $2; }
 | lines \n
 | /* empty */
 | error \n \{ yyerror("reenter last line: ");

; 
...
```

Error production:
set error mode and skip input until newline

Reset parser to normal mode
Emulating the Evaluation of L-Attributed Definitions in Yacc

\[
\begin{align*}
D & \rightarrow T \{ \text{Lin} := T \text{.type} \} \ L \\
T & \rightarrow \text{int} \ {\{ T \text{.type} := \text{‘integer’} \}} \\
T & \rightarrow \text{real} \ {\{ T \text{.type} := \text{‘real’} \}} \\
L & \rightarrow \{ L_1 \text{.in} := L \text{.in} \} \ L_1 , \ id \\
& \{ \text{addtype}(id \text{.entry}, L \text{.in}) \} \\
L & \rightarrow id \ {\{ \text{addtype}(id \text{.entry}, L \text{.in}) \}} \\
\end{align*}
\]

%{
Type Lin; /* global variable */
%
%
D : Ts L
;
Ts : T { Lin = $1; } \\
; \\
T : INT { $$ = TYPE\_INT; } \ |
\text{REAL} { $$ = TYPE\_REAL; } \\
; \\
L : L \text{‘,’} ID { \text{addtype}($3, \text{Lin}); } \ |
\text{ID} { \text{addtype}($1, \text{Lin}); } \\
; \\
%}

27
Rewriting a Grammar to Avoid Inherited Attributes

Production

\[
D \rightarrow L : T \\
T \rightarrow \text{int} \\
T \rightarrow \text{real} \\
L \rightarrow L_1 , \text{id} \\
L \rightarrow \text{id}
\]

Semantic Rule

\[
\text{addtype}(\text{id}.\text{entry}, L.\text{type}) \\
T.\text{type} := \text{‘integer’} \\
T.\text{type} := \text{‘real’} \\
\text{addtype}(\text{id}.\text{entry}, L.\text{type}) \\
L.\text{type} := T.\text{type}
\]