# Context-sensitive Analysis

## Part IV

## *Ad-hoc syntax-directed translation, Symbol Tables, andTypes*

# Quiz

Name two differences between attribute grammars and ad-hoc syntax directed translation techniques?

# Example: Processing C Declarations

| | | |
|---|---|---|
| *DeclarationList* | → | *DeclarationList Declaration* |
| | \| | *Declaration* |
| *Declaration* | → | *SpecifierList InitDeclaratorList* ; |
| *SpecifierList* | → | *Specifier SpecifierList* |
| | \| | *Specifier* |
| *Specifier* | → | *StorageClass* |
| | \| | *TypeSpecifier* |
| *StorageClass* | → | `auto` |
| | \| | `static` |
| | \| | `extern` |
| | \| | `register` |
| *TypeSpecifier* | → | `void` |
| | \| | `char` |
| | \| | `short` |
| | \| | `int` |
| | \| | `long` |
| | \| | `signed` |
| | \| | `unsigned` |
| | \| | `float` |
| | \| | `double` |

# Example: Processing C Declarations

| | | |
|---|---|---|
| *DeclarationList* | → | *DeclarationList Declaration* |
| | \| | *Declaration* |
| *Declaration* | → | *SpecifierList InitDeclaratorList* ; |
| *SpecifierList* | → | *Specifier SpecifierList* |
| | \| | *Specifier* |
| *Specifier* | → | *StorageClass* |
| | \| | *TypeSpecifier* |
| *StorageClass* | → | `auto` |
| | \| | `static` |
| | \| | `extern` |
| | \| | `register` |
| *TypeSpecifier* | → | `void` |
| | \| | `char` |
| | \| | `short` |
| | \| | `int` |
| | \| | `long` |
| | \| | `signed` |
| | \| | `unsigned` |
| | \| | `float` |
| | \| | `double` |

# Example: Processing C Declarations

| | | |
|---|---|---|
| *DeclarationList* | → | *DeclarationList Declaration* |
| | \| | *Declaration* |
| *Declaration* | → | *SpecifierList InitDeclaratorList* ; |
| *SpecifierList* | → | *Specifier SpecifierList* |
| | \| | *Specifier* |
| *Specifier* | → | *StorageClass* |
| | \| | *TypeSpecifier* |
| *StorageClass* | → | `auto` |
| | \| | `static` |
| | \| | `extern` |
| | \| | `register` |
| *TypeSpecifier* | → | `void` |
| | \| | `char` |
| | \| | `short` |
| | \| | `int` |
| | \| | `long` |
| | \| | `signed` |
| | \| | `unsigned` |
| | \| | `float` |
| | \| | `double` |

# Example: Processing C Declarations

| | | |
|---|---|---|
| *DeclarationList* | → | *DeclarationList Declaration* |
| | \| | *Declaration* |
| *Declaration* | → | *SpecifierList InitDeclaratorList* ; |
| *SpecifierList* | → | *Specifier SpecifierList* |
| | \| | *Specifier* |
| *Specifier* | → | *StorageClass* |
| | \| | *TypeSpecifier* |
| *StorageClass* | → | `auto` |
| | \| | `static` |
| | \| | `extern` |
| | \| | `register` |
| *TypeSpecifier* | → | `void` |
| | \| | `char` |
| | \| | `short` |
| | \| | `int` |
| | \| | `long` |
| | \| | `signed` |
| | \| | `unsigned` |
| | \| | `float` |
| | \| | `double` |

# Example: Processing C Declarations

| | | |
|---|---|---|
| *DeclarationList* | → | *DeclarationList Declaration* |
| | \| | *Declaration* |
| *Declaration* | → | *SpecifierList InitDeclaratorList* ; |
| *SpecifierList* | → | *Specifier SpecifierList* |
| | \| | *Specifier* |
| *Specifier* | → | *StorageClass* |
| | \| | *TypeSpecifier* |
| *StorageClass* | → | `auto` |
| | \| | `static` |
| | \| | `extern` |
| | \| | `register` |
| *TypeSpecifier* | → | `void` |
| | \| | `char` |
| | \| | `short` |
| | \| | `int` |
| | \| | `long` |
| | \| | `signed` |
| | \| | `unsigned` |
| | \| | `float` |
| | \| | `double` |

# Example: Processing C Declarations

| | | |
|---|---|---|
| *DeclarationList* | → | *DeclarationList Declaration* |
| | \| | *Declaration* |
| *Declaration* | → | *SpecifierList InitDeclaratorList* ; |
| *SpecifierList* | → | *Specifier SpecifierList* |
| | \| | *Specifier* |
| *Specifier* | → | *StorageClass* |
| | \| | *TypeSpecifier* |
| *StorageClass* | → | `auto` |
| | \| | `static` |
| | \| | `extern` |
| | \| | `register` |
| *TypeSpecifier* | → | `void` |
| | \| | `char` |
| | \| | `short` |
| | \| | `int` |
| | \| | `long` |
| | \| | `signed` |
| | \| | `unsigned` |
| | \| | `float` |
| | \| | `double` |

# Example: Processing C Declarations

| | | |
|---|---|---|
| *InitDeclaratorList* | $\rightarrow$ | *InitDeclaratorList* , *InitDeclarator* |
| | \| | *InitDeclarator* |
| *InitDeclarator* | $\rightarrow$ | *Declarator* = *Initializer* |
| | \| | *Declarator* |
| *Declarator* | $\rightarrow$ | *Pointer DirectDeclarator* |
| | \| | *DirectDeclarator* |
| *Pointer* | $\rightarrow$ | $\star$ |
| | \| | $\star$ *Pointer* |
| *DirectDeclarator* | $\rightarrow$ | `ident` |
| | \| | ( *Declarator* ) |
| | \| | *DirectDeclarator* ( ) |
| | \| | *DirectDeclarator* ( *ParameterTypeList* ) |
| | \| | *DirectDeclarator* ( *IdentifierList* ) |
| | \| | *DirectDeclarator* [ ] |
| | \| | *DirectDeclarator* [ *ConstantExpr* ] |

# Example: Processing C Declarations

| | | |
|---|---|---|
| *InitDeclaratorList* | → | *InitDeclaratorList* , *InitDeclarator* |
| | \| | *InitDeclarator* |
| *InitDeclarator* | → | *Declarator* = *Initializer* |
| | \| | *Declarator* |
| *Declarator* | → | *Pointer DirectDeclarator* |
| | \| | *DirectDeclarator* |
| *Pointer* | → | * |
| | \| | * *Pointer* |
| *DirectDeclarator* | → | `ident` |
| | \| | ( *Declarator* ) |
| | \| | *DirectDeclarator* ( ) |
| | \| | *DirectDeclarator* ( *ParameterTypeList* ) |
| | \| | *DirectDeclarator* ( *IdentifierList* ) |
| | \| | *DirectDeclarator* [ ] |
| | \| | *DirectDeclarator* [ *ConstantExpr* ] |

# Example: Processing C Declarations

| | | |
|---|---|---|
| *InitDeclaratorList* | → | *InitDeclaratorList* , *InitDeclarator* |
| | \| | *InitDeclarator* |
| *InitDeclarator* | → | *Declarator* = *Initializer* |
| | \| | *Declarator* |
| *Declarator* | → | *Pointer DirectDeclarator* |
| | \| | *DirectDeclarator* |
| *Pointer* | → | * |
| | \| | * *Pointer* |
| *DirectDeclarator* | → | `ident` |
| | \| | ( *Declarator* ) |
| | \| | *DirectDeclarator* ( ) |
| | \| | *DirectDeclarator* ( *ParameterTypeList* ) |
| | \| | *DirectDeclarator* ( *IdentifierList* ) |
| | \| | *DirectDeclarator* [ ] |
| | \| | *DirectDeclarator* [ *ConstantExpr* ] |

# Example: Processing C Declarations

| | | |
|---|---|---|
| *InitDeclaratorList* | $\rightarrow$ | *InitDeclaratorList* , *InitDeclarator* |
| | \| | *InitDeclarator* |
| *InitDeclarator* | $\rightarrow$ | *Declarator* = *Initializer* |
| | \| | *Declarator* |
| *Declarator* | $\rightarrow$ | *Pointer DirectDeclarator* |
| | \| | *DirectDeclarator* |
| *Pointer* | $\rightarrow$ | $\star$ |
| | \| | $\star$ *Pointer* |
| *DirectDeclarator* | $\rightarrow$ | `ident` |
| | \| | ( *Declarator* ) |
| | \| | *DirectDeclarator* ( ) |
| | \| | *DirectDeclarator* ( *ParameterTypeList* ) |
| | \| | *DirectDeclarator* ( *IdentifierList* ) |
| | \| | *DirectDeclarator* [ ] |
| | \| | *DirectDeclarator* [ *ConstantExpr* ] |

# Example: Processing C Declarations

| | | |
|---|---|---|
| *InitDeclaratorList* | $\rightarrow$ | *InitDeclaratorList* , *InitDeclarator* |
| | \| | *InitDeclarator* |
| *InitDeclarator* | $\rightarrow$ | *Declarator* = *Initializer* |
| | \| | *Declarator* |
| *Declarator* | $\rightarrow$ | *Pointer DirectDeclarator* |
| | \| | *DirectDeclarator* |
| *Pointer* | $\rightarrow$ | * |
| | \| | * *Pointer* |
| *DirectDeclarator* | $\rightarrow$ | `ident` |
| | \| | ( *Declarator* ) |
| | \| | *DirectDeclarator* ( ) |
| | \| | *DirectDeclarator* ( *ParameterTypeList* ) |
| | \| | *DirectDeclarator* ( *IdentifierList* ) |
| | \| | *DirectDeclarator* [ ] |
| | \| | *DirectDeclarator* [ *ConstantExpr* ] |

## Building a Symbol Table

- Enter declaration information as processed
- At end of declaration syntax, do some post processing
- Use table to check errors as parsing progresses

assumes table is global

# Symbol Table: Typical Uses

- Define before use → lookup on reference
- Dimension, type, ... → check as encountered
- Type checking of expression → bottom-up walk

## Symbol Table: Typical Uses

- Procedure interfaces are harder
  - →Build a representation for parameter list & types
  - →Create list of sites to check

# Is This Really "Ad-hoc" ?

Relationship between practice and attribute grammars

## Similarities

- Both rules & actions associated with productions
- Application order determined by tools, not author

# Is This Really “Ad-hoc” ?

Relationship between practice and attribute grammars

## Differences

- Actions applied as a unit; not true for AG rules
- Anything goes in *ad-hoc* actions; AG rules are functional

# Making Ad-hoc SDT Work

How do we fit this into an LR(1) parser?

*From an earlier lecture*

```
stack.push(INVALID);
stack.push(s0);                     // initial state
token = scanner.next_token();
loop forever {
    s = stack.top();
    if ( ACTION[s,token] == "reduce A→β" ) then {
       stack.popnum(2*|β|);        // pop 2*|β| symbols
        s = stack.top();
        stack.push(A);               // push A
        stack.push(GOTO[s,A]);  // push next state
    }
    else if ( ACTION[s,token] == "shift si" ) then {
        stack.push(token); stack.push(si);
        token ← scanner.next_token();
    }
    else if ( ACTION[s,token] == "accept"
                        & token == EOF )
        then break;
    else throw a syntax error;
}
report success;
```

```
stack.push(INVALID);
stack.push(NULL);
stack.push(s0);                      // initial state
token = scanner.next_token();
loop forever {
    s = stack.top();
    if ( ACTION[s,token] == "reduce A→β" ) then {

        /* insert case statement here */

        stack.popnum(3*|β|);         // pop 3*|β| symbols
        s = stack.top();
        stack.push($$);              // push result
        stack.push(A);               // push A
        stack.push(GOTO[s,A]);       // push next state
    }
    else if ( ACTION[s,token] == "shift si" ) then {
        stack.push(attr); stack.push(token);
        stack.push(si);
        token ← scanner.next_token();
    }
    else if ( ACTION[s,token] == "accept"
                    & token == EOF )
        then break;
    else throw a syntax error;
}
report success;
```

*To add yacc-like actions*

- Stack has 3 items per symbol rather than 2 (3rd is $$)

```
stack.push(INVALID);
stack.push(NULL);
stack.push(s0);                    // initial state
token = scanner.next_token();
loop forever {
    s = stack.top();
    if ( ACTION[s,token] == "reduce A→β" ) then {

        /* insert case statement here */

        stack.popnum(3*|β|);       // pop 3*|β| symbols
        s = stack.top();
        stack.push($$);            // push result
        stack.push(A);             // push A
        stack.push(GOTO[s,A]);     // push next state
    }
    else if ( ACTION[s,token] == "shift si" ) then {
        stack.push(attr); stack.push(token);
        stack.push(si);
        token ← scanner.next_token();
    }
    else if ( ACTION[s,token] == "accept"
                    & token == EOF )
        then break;
    else throw a syntax error;
}
report success;
```

- Add case statement to the reduction processing section
  - → Case switches on production number

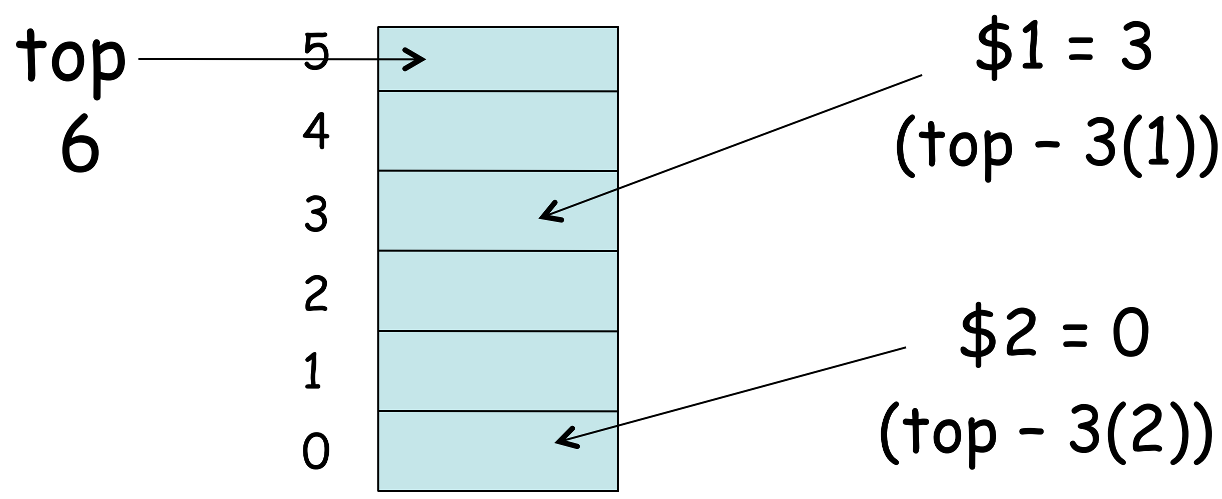
```
stack.push(INVALID);
stack.push(NULL);
stack.push(s0);                          // initial state
token = scanner.next_token();
loop forever {
    s = stack.top();
    if ( ACTION[s,token] == "reduce A→β" ) then {

        /* insert case statement here */

        stack.popnum(3*|β|);       // pop 3*|β| symbols
         s = stack.top();
        stack.push($$);             // push result
         stack.push(A);             // push A
         stack.push(GOTO[s,A]);  // push next state
    }
    else if ( ACTION[s,token] == "shift si" ) then {
        stack.push(attr); stack.push(token);
        stack.push(si);
        token ← scanner.next_token();
    }
    else if ( ACTION[s,token] == "accept"
                    & token == EOF )
        then break;
    else throw a syntax error;
}
report success;
```

- Each case clause holds the code snippet for that production
- Substitute appropriate names for $$, $1, $2, …

# Making Ad-hoc SDT Work

How do we fit this into an LR(1) parser?

- Need a naming scheme to access them
  - → \$n translates into stack location (top - 3n)
- Need to sequence rule applications
  - → On every reduce action, perform the action rule
  - → Add a giant case statement to the parser



# Making Ad-hoc SDT Work

What about a rule that must work in mid-production?

- Can transform the grammar
  - → Split it into two parts at the point where rule must go
  - → Apply the rule on reduction to the appropriate part
- Can also handle reductions on shift actions
  - → Add a production to create a reduction
    - Was: *fee* → *<u>fum</u>*
    - Make it: *fee* → *fie* → *<u>fum</u>*

      and tie the action to this new reduction

Together, these let us apply rule at any point in the parse

## Alternative Strategy

*What if you need to perform actions that do not fit well into the Ad-hoc Syntax-Directed Translation framework?*

- Build the abstract syntax tree using SDT
- Perform the actions during one or more treewalks
  - → In an OOL, think of this problem as a classic application of the visitor pattern
  - → Perform arbitrary computation in treewalk order
  - → Make multiple passes if necessary

*Again, a competent junior or senior CS major would derive this solution after a couple of minutes of thought.*