Computing Inside The Parser
— Syntax-Directed Translation —

Comp 412

Copyright 2017, Keith D. Cooper & Linda Torczon, all rights reserved.
Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.
Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.

Chapter 4 in EaC2e
Where are we? Where are we going?

**In the book:**

- Chapter 3 focused on the membership question
  - Given a grammar $G$, is some sentence $s \in L(G)$?
  - We developed efficient techniques to recognize languages
  - To make useful tools, we need to go well beyond syntax

- Chapter 4 focuses on computing in the parser
  - Strong emphasis on attribute grammars in the written chapter
  - We will ignore that material & take a more ad-hoc, pragmatic approach
  - Lecture will emphasize the applications in compilers ...

- Chapter 5 provides an overview of Intermediate Representations
  - Covered the representation of code in the last two lectures
  - Representation of name spaces coming in the near future

- Chapters 6, & 7 focus on the translation of language features into compiled code — runtime support & code shape
  - We will skip around wildly in these two chapters
Now We Know How To Build Parsers ...

What do we want to do with them?

• Parsers have many applications
  – Obvious answer is “to build IR that will be compiled or interpreted”
  – Reading markup languages: XML, EPUB, GML & KML, RSS and building data structures to represent the content of various ML documents
  – Reading other data representations (e.g., YAML) to transmit data in a rich and structured form
  – Reading in the results of other language processing tools: e.g., EDIF

• We use parsers to understand syntax

General Schema

• Read some input
• Build some data structures to model the input
• Perform some structured computation on the data structures
What else does a compiler need to know about the input program?
• *Short answer:* everything
What else does a compiler need to know about the input program?

• Compiler must assign storage to every item that needs storage
• Compiler must generate code for every construct that executes
• Compiler must plan resource allocation, use, & recycling
• Compiler must ensure coherent use of information (types)

The compiler is guided in this effort by the actual code, the language definition, and its knowledge about the runtime environment.

Meta Issue:

• The compiler does its work at compile time
• The compiler emits code that performs the work at runtime
• Keeping track of the times (design, compile, run) is critical to your understanding of the material
Example: Consistent Use of Values

To ensure consistent use of values, PLs introduce type systems

• Type systems allow the language designer to express constraints that are “deeper” than syntax
• Type systems allow the programmer to write down facts & intentions that are “deeper” than syntax

By comparing constraints against facts & expressed intentions, the compiler can often spot subtle and (otherwise) difficult to find errors

Type Checking

• Requires that the language have a well-designed type system
• Requires that the parser gather some base information on the code
• Requires that the semantic elaborator apply a set of inference rules
  – Inference rules range from simple (C) to complex (ML)
**Example: Overloading of Names**

To allow reuse of individual names, PLs introduce scoping rules

- Procedures start with a clean name space, as do blocks in some PLs
- Programmer can choose names, largely ignoring names used elsewhere
  - Exception: declared names of global values and interfaces
- The part of speech, `identifier`, does not encode enough information for the compiler to generate correct code

Compilers must recognize, model, & understand the scopes in a program

**Compile-time and Runtime support**

- Managing the applications name space requires support during translation (at *compile time*) and during execution (at *runtime*)
- Compilers build tables to encode knowledge at compile time
- Compilers emit code to build, maintain, & use runtime structures that support proper name resolution at runtime
Example: Storage Layout

To access data, running code needs an address in runtime RAM

• Running program cannot issue a load or a branch without an address
• Every item of code or data requires an address
  – Addresses are in the virtual address space of a new process
  – **Meta Issue:** that process won’t exist until a user runs the compiled code
  – **Meta Issue:** code may run on different hardware and OS versions

The compiler must manage all decisions about storage layout, for the entire program (not just one procedure).

Storage Layout

• Semantic elaborator must lay out storage before it generates code
• For each item of code or data, the compiler must either:
  – Assign a symbolic address to each item, *or*
  – Implement a scheme whereby its address can be computed
Example: Separate Compilation

Programmers write code in modules & combine module into programs
- Compiler typically sees one module, or file, at a time
- Compilation for a module is, largely, context free
- Compiled code must link with other pre-compiled modules and/or pre-compiled library routines & system calls

Planning
- The compiler-writer must plan & standardize the ways in which the compiler will translate the source code, to create standard conventions for naming code & data, & for calling other procedures
  - Conventions for naming code and data segments
  - Conventions for calls & returns, for memory use, for system calls, ...
- The compiler must implement those plans carefully & completely
Syntax-Directed Translation

All of these issues play into SDT

- Answers depend on computation over values, not parts of speech
- Questions & answers involve non-local information
- Questions and answers are “deeper” than syntax

How can we answer these questions?

- Use formal methods
  - Attribute grammars, rule-based systems
- Use *ad-hoc* techniques
  - Symbol tables, *ad-hoc* code

*Formalisms work well for scanning & parsing. Real compilers use them. For these “context-sensitive” issues, ad-hoc techniques dominate practice.*
Example

Computing the value of an unsigned integer

Consider the simple grammar

```
1  Number    \rightarrow digit DigitList
2  DigitList \rightarrow digit DigitList
3              | epsilon
```

One obvious use of the grammar is to convert an ASCII string of the number to its integer value

• Build computation into parser
• An easy intro to syntax-directed translation
Example

Computing the value of an unsigned integer

Consider the simple grammar

\[
\begin{align*}
1 & \quad Number \rightarrow \text{digit DigitList} \\
2 & \quad DigitList \rightarrow \text{digit DigitList} \\
3 & \quad | \quad \text{epsilon}
\end{align*}
\]

One obvious use of the grammar is to convert an ASCII string of the number to its integer value

- Build computation into parser
- An easy intro to syntax-directed translation

and its recursive-descent parser

```java
boolean Number( ) {
    if (word = digit) then {
        word = NextWord( );
        return DigitList( );
    }
    else return false;
}

boolean DigitList( ) {
    if (word = digit) then {
        word = NextWord( );
        return DigitList( );
    }
    else return true; /* epsilon case */
}
```

This example is for illustrative purposes. You don’t need a CFG to recognize integers.
Example

Computing the value of an unsigned integer

Consider the simple grammar

\[
\begin{align*}
1 & \quad Number \rightarrow \text{digit} \ DigitList \\
2 & \quad DigitList \rightarrow \text{digit} \ DigitList \\
3 & \quad | \quad \epsilon
\end{align*}
\]

Any assembly programmer will tell you to accumulate the value, left to right, multiplying by the left-context value by 10 at each step.

- e.g., \(147 = (1 \times 10 + 4) \times 10 + 7\)

and to get the value of character \(d\) by computing \(d - '0'\)

The programming "trick", however, is undoubtedly how \texttt{atoi()} works.

```java
boolean Number( ) {
    if (word = digit) then {
        word = NextWord( );
        return DigitList( );
    }
    else return false;
}

boolean DigitList( ) {
    if (word = digit) then {
        word = NextWord( );
        return DigitList( );
    }
    else return true; /* epsilon case */
}
```
Example

Computing the value of an unsigned integer

```
boolean Number( ) {
    if (word = digit) then {
        word = NextWord( );
        return DigitList( );
    }
    else return false;
}

boolean DigitList( ) {
    if (word = digit) then {
        word = NextWord( );
        return DigitList( );
    }
    else return true; /* epsilon case */
}
```

The Plan

1. Make Number() and DigitList() take value as an argument and return a (boolean,value) pair
Example

Computing the value of an unsigned integer

boolean Number( ) {
    if (word = digit) then {
        word = NextWord( );
        return DigitList( );
    }
    else return false;
}

boolean DigitList( ) {
    if (word = digit) then {
        word = NextWord( );
        return DigitList( );
    }
    else return true; /* epsilon case */
}

The Plan
1. Make Number() and DigitList() take value as an argument and return a (boolean,value) pair
2. Compute initial digit in Number()
Example

Computing the value of an unsigned integer

```java
boolean Number( ) {
    if (word = digit) then {
        word = NextWord( );
        return DigitList( );
    }
    else return false;
}

boolean DigitList( ) {
    if (word = digit) then {
        word = NextWord( );
        return DigitList( );
    }
    else return true; /* epsilon case */
}
```

The Plan

1. Make Number() and DigitList() take value as an argument and return a (boolean,value) pair
2. Compute initial digit in Number()
3. Add second & subsequent digits in DigitList()
Example

Computing the value of an unsigned integer

pair (boolean, int) Number( value ) {
    if (word = digit) then {
        value = ValueOf( digit);
        word = NextWord( );
        return DigitList( value );
    }
    else return (false, invalid value);
}

pair (boolean, int) DigitList( value ) {
    if (word = digit) then {
        value = value * 10 + ValueOf( digit);
        word = NextWord( );
        return DigitList( value );
    }
    else return (true, value);
}

The Plan

1. Make Number() and DigitList() take value as an argument and return a (boolean,value) pair
2. Compute initial digit in Number()
3. Add second & subsequent digits in DigitList()

Obvious, invented syntax for a pair
Example

Computing the value of an unsigned integer

Consider the simple grammar

1 \[ Number \rightarrow \text{digit } DigitList \]
2 \[ DigitList \rightarrow \text{digit } DigitList \]
3 \[ | \epsilon \]

Ok, so it works with recursive descent.

What about an LR(1) parser?
Example

Computing the value of an unsigned integer

Consider the left-recursive grammar

1. \( Number \rightarrow DigitList \)
2. \( DigitList \rightarrow DigitList \ digit \)
3. \( \mid digit \)

We would like to augment the grammar with actions that compute the value of the number

- Cannot encode this computation into the context-free syntax
- Relies on the lexeme of each digit, rather than its part of speech
## Example

### Parse the number 976

<table>
<thead>
<tr>
<th>State</th>
<th>Lookahead</th>
<th>Stack</th>
<th>Handle</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td>digit (9)</td>
<td>$0</td>
<td>—none—</td>
<td>—</td>
</tr>
<tr>
<td>0</td>
<td>digit (9)</td>
<td>$0</td>
<td>—none—</td>
<td>shift 2</td>
</tr>
<tr>
<td>2</td>
<td>digit (7)</td>
<td>$0 9 2</td>
<td>DL → digit</td>
<td>reduce 3</td>
</tr>
<tr>
<td>1</td>
<td>digit (7)</td>
<td>$0 DL 1</td>
<td>—none—</td>
<td>shift 3</td>
</tr>
<tr>
<td>3</td>
<td>digit (6)</td>
<td>$0 DL 17 3</td>
<td>DL → DL digit</td>
<td>reduce 2</td>
</tr>
<tr>
<td>1</td>
<td>digit (6)</td>
<td>$0 DL 1</td>
<td>—none—</td>
<td>shift 3</td>
</tr>
<tr>
<td>3</td>
<td>EOF</td>
<td>$0 DL 16 3</td>
<td>DL → DL digit</td>
<td>reduce 2</td>
</tr>
<tr>
<td>1</td>
<td>EOF</td>
<td>$0 DL 1</td>
<td>Number → DL</td>
<td>accept</td>
</tr>
</tbody>
</table>

### ACTION

<table>
<thead>
<tr>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>s 2</td>
<td>1</td>
</tr>
<tr>
<td>s 3</td>
<td>acc</td>
</tr>
<tr>
<td>r 2</td>
<td></td>
</tr>
</tbody>
</table>

### GOTO

<table>
<thead>
<tr>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>DigitList</td>
</tr>
</tbody>
</table>

Notice that it reduces the 9 with $\text{DL} \rightarrow \text{digit}$, and the others with $\text{DL} \rightarrow \text{DL digit}$.

*Rightmost derivation in reverse!*
Example

Parse the number 976

<table>
<thead>
<tr>
<th>State</th>
<th>Lookahead</th>
<th>Stack</th>
<th>Handle</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td>digit (9)</td>
<td>$ 0</td>
<td>—none—</td>
<td>—</td>
</tr>
<tr>
<td>0</td>
<td>digit (9)</td>
<td>$ 0</td>
<td>—none—</td>
<td>shift 2</td>
</tr>
<tr>
<td>2</td>
<td>digit (7)</td>
<td>$ 0 9 2</td>
<td>DL $\to$ digit</td>
<td>reduce 3</td>
</tr>
<tr>
<td>1</td>
<td>digit (7)</td>
<td>$ 0 DL 1</td>
<td>—none—</td>
<td>shift 3</td>
</tr>
<tr>
<td>3</td>
<td>digit (6)</td>
<td>$ 0 DL 1 7 3</td>
<td>DL $\to$ DL digit</td>
<td>reduce 2</td>
</tr>
<tr>
<td>1</td>
<td>digit (6)</td>
<td>$ 0 DL 1</td>
<td>—none—</td>
<td>shift 3</td>
</tr>
<tr>
<td>3</td>
<td>EOF</td>
<td>$ 0 DL 1 6 3</td>
<td>DL $\to$ DL digit</td>
<td>reduce 2</td>
</tr>
<tr>
<td>1</td>
<td>EOF</td>
<td>$ 0 DL 1</td>
<td>Number $\to$ DL</td>
<td>accept</td>
</tr>
</tbody>
</table>

Two cases that correspond to the actions in our recursive descent parser

- The leftmost digit should just set value to $\text{ValueOf}(\text{digit})$
- Subsequent digits should compute $\text{value} \times 10 + \text{ValueOf}(\text{digit})$

Suggests that we perform the calculations when the parser reduces
### Example

**Parse the number 976**

<table>
<thead>
<tr>
<th>State</th>
<th>Lookahead</th>
<th>Stack</th>
<th>Handle</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td>digit (9)</td>
<td>$ 0</td>
<td>—<em>none</em>—</td>
<td>—</td>
</tr>
<tr>
<td>0</td>
<td>digit (9)</td>
<td>$ 0</td>
<td>—<em>none</em>—</td>
<td>shift 2</td>
</tr>
<tr>
<td>2</td>
<td>digit (7)</td>
<td>$ 0 9 2</td>
<td><em>DL</em> → <em>digit</em></td>
<td>reduce 3</td>
</tr>
<tr>
<td>1</td>
<td>digit (7)</td>
<td>$ 0 DL 1</td>
<td>—<em>none</em>—</td>
<td>shift 3</td>
</tr>
<tr>
<td>3</td>
<td>digit (6)</td>
<td>$ 0 DL 1 7 3</td>
<td><em>DL</em> → <em>DL</em> <em>digit</em></td>
<td>reduce 2</td>
</tr>
<tr>
<td>1</td>
<td>digit (6)</td>
<td>$ 0 DL 1</td>
<td>—<em>none</em>—</td>
<td>shift 3</td>
</tr>
<tr>
<td>3</td>
<td>EOF</td>
<td>$ 0 DL 1 6 3</td>
<td><em>DL</em> → <em>DL</em> <em>digit</em></td>
<td>reduce 2</td>
</tr>
<tr>
<td>1</td>
<td>EOF</td>
<td>$ 0 DL 1</td>
<td><em>Number</em> → <em>DL</em></td>
<td>accept</td>
</tr>
</tbody>
</table>

### Computing on reductions

- Reduction by production 3 should set `value` to `ValueOf(digit)`
- Reduction by 2 should compute `value = value * 10 + ValueOf(digit)`
Example

Performing calculations on the reduce actions

<table>
<thead>
<tr>
<th>State</th>
<th>Lookahead</th>
<th>Stack</th>
<th>Calculation</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td>digit (9)</td>
<td>$ 0</td>
<td></td>
<td>—</td>
</tr>
<tr>
<td>0</td>
<td>digit (9)</td>
<td>$ 0</td>
<td></td>
<td>shift 2</td>
</tr>
<tr>
<td>2</td>
<td>digit (7)</td>
<td>$ 0 9 2</td>
<td>[value \leftarrow 9]</td>
<td>reduce 3</td>
</tr>
<tr>
<td>1</td>
<td>digit (7)</td>
<td>$ 0 DL 1</td>
<td></td>
<td>shift 3</td>
</tr>
<tr>
<td>3</td>
<td>digit (6)</td>
<td>$ 0 DL 1 7 3</td>
<td>[value \leftarrow value \times 10 + 7]</td>
<td>reduce 2</td>
</tr>
<tr>
<td>1</td>
<td>digit (6)</td>
<td>$ 0 DL 1</td>
<td></td>
<td>shift 3</td>
</tr>
<tr>
<td>3</td>
<td>EOF</td>
<td>$ 0 DL 1 6 3</td>
<td>[value \leftarrow value \times 10 + 6]</td>
<td>reduce 2</td>
</tr>
<tr>
<td>1</td>
<td>EOF</td>
<td>$ 0 DL 1</td>
<td></td>
<td>accept</td>
</tr>
</tbody>
</table>

This style of computation is called Ad Hoc Syntax-Directed Translation

• Compiler writer provides code snippets for specific productions
• Parser actions determine specific actions & the overall sequence
SDT in Bison or Yacc

We specify SDT actions using a simple notation

\[
\begin{align*}
1 & \quad Number & \rightarrow & \quad DigitList & \quad \{$$ = $1; \}\}
2 & \quad DigitList & \rightarrow & \quad DigitList \ digit & \quad \{$$ = $1 \times 10 + $2; \}\}
3 & \quad | & \quad digit & \quad \{$$ = ValueOf(\text{digit}); \}\}
\end{align*}
\]

In Bison and Yacc, the compiler writer provides production-specific code snippets that execute when the parser reduces by that production

• Positional notation for the value associated with a symbol
  – \$$ is the LHS; $1$ is the first symbol in the RHS; $2$ the second, ...

• Compiler writer can put arbitrary code in these snippets
  – Solve a travelling salesman problem, compute PI to 100 digits, ...
  – More importantly, they can compute on the lexemes of grammar symbols and on information derived and stored earlier in translation
Ad Hoc Syntax-Directed Translation

Why do we call this “ad hoc” syntax-directed translation?

• We have a formalism to specify syntax-directed translation schemes
  – Attribute grammars (see EaC2e, § 4.3)
  – An attribute is a value associated with an instance of a grammar symbol
    → Associated with a node in the parse tree

• In formalism, compiler writer creates a specification & tools generate the code that performs the actual computation
  – Attribute-grammar evaluator generator (similar to parser generator)
  – Syntax-directed because the specification is written in terms of the grammar and the corresponding parse tree (or syntax tree)
  – Evaluator takes an “unattributed” tree and produces an “attributed tree”

• Attribute grammar systems have strengths & weaknesses
  – They have never quite become popular
  – If you are interested, Section 4.3 in EaC2e is a primer

We will focus on ad hoc SDT schemes. They are widely used in practice.
Actions are taken on reductions

- Insert a call to `Work()` before the call to `stack.popnum()`

- `Work()` contains a case statement that switches on the production number

- Code in `Work()` can read items from the stack

  → That is why it calls `Work()` before `stack.popnum()`
Fitting AHSĐT into the **LR(1)** Skeleton Parser

**Passing values between actions**

- Tie values to instances of grammar symbols
  - Equivalent to parse tree nodes
- We can pass values on the stack
  - Push / pop 3 rather than 2
  - *Work()* takes the stack as input (*conceptually*) and returns the value for the reduction it processes
  - *Shift* creates initial values

```plaintext
stack.push(INVALID);
stack.push(s₀); // initial state
word = scanner.next_word();
loop forever {
    s = stack.top();
    if ( ACTION[s,word] == “reduce A→β” ) then {
        stack.popnum(2*|β|); // pop 2*|β| symbols
        s = stack.top();
        stack.push(A); // push LHS, A
        stack.push(GOTO[s,A]); // push next state
    } else if ( ACTION[s,word] == “shift sᵢ” ) then {
        stack.push(word); stack.push(sᵢ);
        word ← scanner.next_word();
    } else if ( ACTION[s,word] == “accept” & word == EOF )
        then break;
    else throw a syntax error;
} report success;
```
Fitting AHS DT into the LR(1) Skeleton Parser

- Modifications are minor
  - Insert call to `Work()`
  - Change the push() & pop() behavior
- Same asymptotic behavior as the original algorithm.
  - 50% more stack space
- Last obstacle is making it easy to write the code for `Work()`

Note that, in C, the stack has some odd union type.
Translating Code Snippets Into *Work()*

For each production, the compiler writer can provide a code snippet

```
{ value = value * 10 + digit; }
```

We need a scheme to name stack locations. Yacc introduced a simple one that has been widely adopted.

- `$$` refers to the result, which will be pushed on the stack
- `$1` is the first item on the productions right hand side
- `$2` is the second item
- `$3` is the third item, and so on ...

The digits example above becomes

```
{ $$ = $1 * 10 + $2; }
```
Translating Code Snippets Into `Work()`

**How do we implement `Work()`?**

- `Work()` takes 2 arguments: the stack and a production number
- `Work()` contains a case statement that switches on production number
  - Each case contains the code snippet for a reduction by that production
  - The $1$, $2$, $3$ ... macros translate into references into the stack
  - The $$ macro translates into the return value

```plaintext
... if ( ACTION[s,word] == "reduce A→β" ) then {
    r = Work(stack, "A→β")
    stack.popnum(3*|β|); // pop 3*|β| symbols
    s = stack.top(); // save exposed state
    stack.push(A); // push A
    stack.push (r); // push result of WORK()
    stack.push(GOTO[s,A]); // push next state
}...
```

$$ translates to r  
$i$ translates to the stack location $3 * ( |β| - i + 1)$ units down from stacktop

Note that $β$, $i$, $3$, and $1$ are all constants so $i$ can be evaluated to a compile-time constant