Syntax-Driven Translation, III
Comp 412

Copyright 2019, 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.

Chapters 4, 5, 6 & 7 in EaC2e
The compiler needs to translate the input program into something that can be executed.

- The compiler writer plans how the pieces get translated,
- how the pieces fit together and play together,
- and how to make all of that efficient.

And the story involves design time, compile time, & runtime.

The Plan

- First, discuss syntax-directed translation
  - Computation specified with the grammar
  - More complex examples with complex information flow
- Then, discuss name spaces & storage layout
- Finally, talk about code shape for calls and control-flow operations
Names, Types, Scopes, Dimensions & Location

For each identifier and each literal constant, the compiler must know:

• Its type, size, & lifetime (e.g., block, procedure, file, or entire program)
• Its dimension (scalar, structure, object, array, array of structure, …)
• Its location (register, address in RAM, offset from some knowable address)

Most of that information can be derived from declarations

• Programmer specifies her intent for each named identifier in the code
• Literal constants can be inferred from syntax and context

```
Decl → Type NameList ;
Type → INT
  | CHAR
NameList → NameList , NAME
  | NAME
```

Fragment of typical grammar

• Datatype is defined in the production for Type
• Datatype for a name is set in the production(s) for Spec
• SDT action must carry value from Type to NAME
Limitations of SDT

Consider the following declaration

```
int x, y, z;
```

The compiler needs to get \textit{INT} from the \textit{Type} node to the \textit{Name} nodes

- Propagation across long distances
- Propagation both up and down in the syntax tree

\textbf{SDT} rules don’t model this kind of value propagation well.
Consider the following declaration

```c
int x, y, z;
```

We can add action rules:

```
Decl  →  Type NameList ;
Type  →  INT  { $$ ← INT; } |
       CHAR { $$ ← CHAR; }
NameList → NameList , NAME |
         NAME
```

But those rules cannot get the type down the NameList structure to the `NAME`s.
Limitations of **SDT**

Consider the following declaration

\[ \text{int } x, y, z; \]

We need to reach outside the SDT framework.

- **SDT** only provides local names (\$$, \$1, \$2, \ldots\$$)
- We need to pass values from one rule to another
- Use a global variable, **CUR**

Framework passes values up the tree. Reach outside to pass down or across the tree.
Limitations of SDT

This technique is ugly, but effective

Consider the following declaration

```
int x, y, z;
```

Using an outside variable, CUR

```
Decl → Type SpecList ;
Type → INT { CUR ← INT; }
   | CHAR { CUR ← CHAR; }
SpecList → SpecList , Spec
   | Spec
Spec → NAME
   { set type of NAME to CUR;1 }
```

1 In the compiler’s symbol table
Another way to handle this kind of problem is to refactor the grammar

- We tend to think of the grammar from the language viewpoint
  → *What syntax does it generate?*

- We can think of grammar as a computational vehicle
  → *What does it allow us to compute?*

\[
\begin{align*}
Decl & \quad \rightarrow \quad \text{INT} \quad \text{IntList} ; \\
& \quad \quad \quad \quad \quad \quad | \quad \text{CHAR} \quad \text{CharList} ; \\
IntList & \quad \rightarrow \quad IntList , \quad \text{NAME} \\
CharList & \quad \rightarrow \quad \text{NameList} , \quad \text{NAME} \\
\end{align*}
\]
Another way to handle this kind of problem is to refactor the grammar.

- We tend to think of the grammar from the language viewpoint → *What syntax does it generate?*
- We can think of grammar as a computational vehicle → *What does it allow us to compute?*

The type is implicit in the rule:
- Actions can use type as a constant
- Avoids opaque use of a global
- Must still record the types in the symbol table

The grammar is shaped for the computation, not the definition:
- Leads to cleaner, more obvious, more maintainable code
A More Complex Example

Generating code for an if-then-else construct

\[
Stmt \rightarrow \text{IF} \ LPAREN \ Expr \ RPAREN \ \text{THEN} \ WithElse \ \text{ELSE} \ WithElse \\
| \quad ... \\
Expr \rightarrow \quad ... \\
WithElse \rightarrow \quad ...
\]

- Control-flow constructs require branches and labels
- Need a schema for how to implement an if-then-else
  - Evaluate the expression
  - Based on its value, branch to the then part or the else part
  - After evaluating the appropriate part, branch to the next statement
    - \textit{We will call that the point in the code the “exit” to simplify talking about it}

\textit{We will assume that the grammar has Boolean & relational expressions}

(Fig. 7.7, p 351 in EaC2e)
## Boolean & Relational Expressions

First, we need to add boolean & relational expressions to the grammar

<table>
<thead>
<tr>
<th>Boolean</th>
<th>$\rightarrow$</th>
<th>Boolean $\lor$ AndTerm</th>
<th>Expr</th>
<th>$\rightarrow$</th>
<th>Expr $+$ Term</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>AndTerm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AndTerm</td>
<td>$\rightarrow$</td>
<td>AndTerm $\land$ RelExpr</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>RelExpr</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RelExpr</td>
<td>$\rightarrow$</td>
<td>RelExpr $&lt;$ Expr</td>
<td>Term</td>
<td>$\rightarrow$</td>
<td>Term $\times$ Value</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RelExpr $\leq$ Expr</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>RelExpr $=$ Expr</td>
<td>Value</td>
<td>$\rightarrow$</td>
<td>! Factor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RelExpr $\neq$ Expr</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>RelExpr $\geq$ Expr</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>RelExpr $&gt;$ Expr</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Expr</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This allows $w < x < y < z$

... where Reference derives a name, a subscripted name, a structure reference, a string reference, a function call, ...
A More Complex Example

Generating code for an if-then-else construct

```
Stmt  →  IF LPAREN Expr RPAREN THEN WithElse ELSE WithElse
       |  ...
Expr  →  ...
WithElse  →  ...
```

- Control-flow constructs require branches and labels
- To generate code with SDT, need to create & track the labels
  1. Create labels for “then part”, “else part”, and the “exit”
  2. Emit branch to appropriate part (then or else) after RPAREN
  3. Emit label for then part before first WithElse
  4. At end of WithElse, emit branch to the exit
  5. Emit label for else part before second WithElse
  6. At end of second WithElse, emit branch to exit
  7. Emit label for the exit on a nop after the second WithElse
A More Complex Example

Generating code for an if-then-else construct

We need a way to hang code snippets in the middle of the RHS

→ Use an old trick; insert an epsilon production where we want to reduce

\[
Stmt \rightarrow \text{IF LPAREN Expr RPAREN CreateAndBranch THEN EmitThenLabel WithElse EmitExitJump ELSE EmitElseLabel WithElse EmitExitJump EmitExitLabel}
\]

\[
\begin{align*}
Stmt & \rightarrow \text{IF LPAREN Expr RPAREN CreateAndBranch} \\
& \quad \text{THEN EmitThenLabel WithElse EmitExitJump} \\
& \quad \text{ELSE EmitElseLabel WithElse EmitExitJump} \\
& \quad \text{EmitExitLabel}
\end{align*}
\]

\[
\begin{align*}
CreateAndBranch & \rightarrow \varepsilon \\
EmitThenLabel & \rightarrow \varepsilon \\
EmitExitJump & \rightarrow \varepsilon \\
EmitElseLabel & \rightarrow \varepsilon \\
EmitExitLabel & \rightarrow \varepsilon
\end{align*}
\]
A More Complex Example

Generating code for an if-then-else construct

• To generate code with SDT, need to create & track the labels
  1. Create labels for “then part”, “else part”, and “exit”
     → Need a structure to hold the three labels
     → Generate three labels and push them onto the stack in CreateAndBranch action
  2. Emit branch to appropriate part (then or else) after RPAREN
     → Emit the branch in CreateAndBranch action
  3. Emit label for then part before first WithElse
     → Emit a labelled nop in EmitThenLabel action
  4. At end of WithElse, emit branch to the exit
     → Emit a jump to the exit label in EmitExitLabel action
  5. Emit label for else part before second WithElse
     → Emit a labelled nop in EmitElseLabel action
  6. At end of second WithElse, emit branch to the exit
     → Handled by 4. above
  7. Emit label for the exit on a nop after the second WithElse
     → Emit a labelled nop in EmitExitLabel action

Could combine these two
A More Complex Example

**Bison supports this idea**

- Allows code snippets between any two grammar symbols on the RHS of a production
- Generates the appropriate epsilon production, its reduction, and ties the action to this new reduction
  - Actions work in the name space of the production where they are written
  - Allows the notation to handle these code snippets in a natural way

**Sample from the compiler for DEMO**

ITE: IF LP Bool RP

```plaintext
{
  $$$ = new ITEHEAD ;
  $$$->then_label = NextLabel;
  $$$->else_label = NextLabel;
  $$$->exit_label = NextLabel;
  emit_branch($$->reg, $$$->then_label,
             $$$->else_label);
  emit_nop($$->then_label,"then part");
}
THEN WithElse
{
  emit_jump($$->exit_label);
  emit_nop($$->else_label);
}
ELSE WithElse
{
  emit_jump($$->exit_label);
  emit_nop($$->exit_label);
}
```

Combines several items into one action

This does get tricky. Bison introduces the ε-productions, which can introduce new shift-reduce conflicts and changes the numbering of the $i$ macros. (See the Bison manual.)
Naming

Before the compiler can translate expressions and statements, it must know, for each name, what that name represents & where it lives.

• The compiler uses a set of symbol tables to model the name space
  – At each point, it has a collection of tables & search paths
  – A path for lexical scoping and a path for the inheritance hierarchy
  – Builds the tables and scopes, with SDT, as it parses the source code

• Lexical scopes
  – Compiler creates a table as it enters a new scope
  – Creates a search path for each scope that models lexical naming

• Inheritance hierarchy
  – Compiler creates a table as it enters a new class definition
  – Creates a search path for that class that monitors inheritance

*Enters is a parse-time event, specified in the grammar (SDT)*
Lexical Scope Example

Path: Fee
Scope Fee:
integer a;
character b, c;

Scope Foe:
integer c;
character d;

Scope: Foe
Path: Foe, Fee

Path: Foe

Scope: Fee
Path: Fee

Scope: Fum
Path: Fum, Fee

Scope: Fum
float b;
boolean d;

Symbol Tables
Path can be represented with a list of tables or a chain of pointers.
Class Definitions

Inheritance Example

Class Point {
  public int x, y;
  private int x;
  public void draw() { ... };
  public void move() { ... };
}

Path: Point, Class

Class CPoint extends Point {
  private color c;
  public void draw() { ...};
  public void setc( Color x )
    { this.c = x; };
}

Path: Cpoint, Point, Class

Inheritance Tables

Class: Class
Superclass: none
Internals do not matter

Class: Point
Superclass: Class

<table>
<thead>
<tr>
<th>Type</th>
<th>Visibility</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>pub</td>
<td>x</td>
</tr>
<tr>
<td>int</td>
<td>pub</td>
<td>y</td>
</tr>
<tr>
<td>int</td>
<td>pri</td>
<td>z</td>
</tr>
<tr>
<td>void()</td>
<td>pub</td>
<td>draw</td>
</tr>
<tr>
<td>void()</td>
<td>pub</td>
<td>move</td>
</tr>
</tbody>
</table>

Class: CPoint
Superclass: Point

<table>
<thead>
<tr>
<th>Method</th>
<th>Visibility</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Color</td>
<td>pri</td>
<td>c</td>
</tr>
<tr>
<td>void()</td>
<td>pub</td>
<td>draw</td>
</tr>
<tr>
<td>void()</td>
<td>pub</td>
<td>setc</td>
</tr>
</tbody>
</table>
Compile-Time Name Resolution

**In the parser:**
- The parser has the lexeme
- Parser performs a lookup in the appropriate search path
  - Syntax may indicate some specific scope
  - Location in the parse provides the default scope
- Lookup maps the lexeme to a specific table record
- Store reference to record in the IR

**Later in compilation:**
- Use the reference from the IR and avoid the search

**The table record records all the relevant attributes**
- Type, size, dimension, storage class, *location*, and *access method* ...
What About Storage Layout?

Where does all of this stuff go? And how does it get there?

The compiler must classify all code & data so that it can assign storage

- **Scopes**: local, global, subject to some set of lexical scoping constraints
- **Lifetimes**: entire execution, execution of a procedure, or indeterminate

Given these classifications & the state of the naming model, the compiler can assign data to specific data areas

- Each procedure has a local data area
  - Local data from a procedure scope or sub-scope usually goes here
- Declarations, procedures, files, & modules may have a static data area
  - Depends on the specific declarations in the code
- A program may have one or more global data areas & constant pools
- Each object has an object record
  - A class, being an object, has an object record of class “class”
When Does the Compiler Assign Storage?

The compiler must assign storage before using addresses in the IR

• Compiler may assign storage before generating initial IR
  – Implies a declaration before use rule or two passes (which requires IR)
• Compiler may assign storage incrementally as it generates IR
  – Leads to some inefficiency in memory layout
    → In practice, probably not too terribly wasteful
• Compiler may generate initial IR with abstracted addresses
  – See @x in any ILOC example with a loadAI
• Any of these schemes can work in the compiler

The PL designer makes decisions that can dictate the choice

• “declare before use” rule is fairly common
• No declarations generally force the second or third option
When Does the Compiler Assign Storage?

With declare before use:  

- Compiler can make its decisions after parsing declarations
- Needs a time and place to perform storage layout

### Typical Grammar for a Procedure

- **Procedure** → Header \{ Declaractions Statements \}
- **Declarations** → Declarations Decl | Decl
- **Statement** → Statements Stmt | Stmt

Modify the grammar to create a reduction to assign storage.

- **Procedure** → Header \{ AssignedStorage Statement \}
- **AssignedStorage** → Declarations
The compiler must assign a location to each named variable and each temporary value that it introduces? Where do all these values live?

• The answer should depend on the value’s visibility & its lifetime

Lifetimes

Automatic variables

• Implicit allocation & deallocation
• Lifetime matches lifetime of declaring scope’s activation

Static variables

• Implicit allocation and deallocation
• Lifetime is the complete execution of the program

Values with irregular lifetimes

• Explicit allocation and deallocation (new, or malloc, or …)
Storage Layout

The compiler must assign a location to each named variable and each temporary value that it introduces? Where do all these values live?

• The answer should depend on the value’s visibility & its lifetime

Visibility

Local Scopes
• Procedure scope: name is visible in defining procedure (& nested ones)
• Block scope (C, scheme): name is visible in defining block (& nested ones)

Global Scope
• Names are visible anywhere that they are declared, unless obscured by a local declaration

Unusual scopes
• C’s “file scope”; languages with explicit visibility control e.g., public or private
Where do all these values live? AR refers to a procedure’s activation record

Automatic & Local

- Automatic local values live in the local data area, in the procedure’s AR
  → Or in a register, if it is an unambiguous scalar value
- Compiler lays out a map for the local data area, including spill locations
- Local values from surrounding scopes live in the AR for their defining scope
  → Runtime addressability mechanism establishes a way to reach them

Static (implies lifetime of entire execution)

- Procedure static ⇒ procedure-specific data area (e.g., &procname_da)
- File static ⇒ file-specific data area (e.g., &filename_da)
- Necessary to ensure that they remain live across different activations

Global (implies lifetime of entire execution)

- Global values live in one or more global data area(s)
- One label per variable, or per file, or per program, ...
  → With consistently mangled labels to connect them across compiles
  → Equivalent to an assembly language “defined storage” pseudo-operation

Compilers obtain these labels by “mangling” the original name. The linker will report conflicting definitions of the same label.
**Activation Record Basics**

In most systems, Activation Records have a similar layout:

<table>
<thead>
<tr>
<th>ARP</th>
<th>Space for parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>k</td>
<td>register save area</td>
</tr>
<tr>
<td></td>
<td>slot for return value</td>
</tr>
<tr>
<td></td>
<td>slot for return address</td>
</tr>
<tr>
<td></td>
<td>slot for addressability</td>
</tr>
<tr>
<td></td>
<td>slot for caller’s ARP</td>
</tr>
<tr>
<td></td>
<td>space for local variables</td>
</tr>
</tbody>
</table>

**ARP** ≡ **Activation Record Pointer**

- Space for parameters to the current procedure
- Contents of saved registers
- If function, space for return value
- Address to resume caller
- Help with non-local access
- To restore caller’s AR on return
- Space for local variables, temporaries, & spills

If ARs are stack allocated, the stack grows in this direction:

- One AR for each invocation of a procedure
- One register dedicated to hold the current ARP

---

What is this “activation record”?
Name Mangling

One of the most useful tools the compiler has is an assembly-code label

- Compiler can emit a label in the code
- Assembler will turn it into a relocatable symbol
- Linker will replace the label with the appropriate virtual address
- Hardware will translate virtual address to a physical address

Name mangling is the standard trick

- The compiler assumes that some source-language names are unique
  - Global variables, procedure names
  - Use fully-qualified names for nested objects
- The compiler builds strings based on the unique name
  - Code for procedure fee might be labelled fee or &fee or ...
  - Specific “mangles” for specific purposes
    → Static data area, entry point to the code, return address at a call, ...
  - Use character combinations that are not legal in the source language
Alignment Issues: One More Standard Trick

Having values of multiple types in the same data area creates potential alignment issues

• Each type (size) of value has its own alignment restriction
• Laying out a data area in arbitrary order can require padding
  – Consider \(a\) & \(c\) as single-byte characters and \(b\) & \(d\) as single-word integers

A Layout That Wastes Space

A Better Layout
Alignment Issues: One More Standard Trick

To create a layout that minimizes space wasted in “padding”

• Sort variables by alignment restrictions
  – Most strict (e.g., quad word or double word) to least strict (e.g., byte)

• From most strict to least strict, lay out variables in contiguous memory
  – Alignment constraints generally decrease by ½ at a transition
  – This scheme avoids almost all padding for scalar variables.
  – Structures that have atypical alignment constraints may need padding

→ A seventeen-byte long structure followed by a quadword aligned double double

```
<table>
<thead>
<tr>
<th>b</th>
<th>d</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

whole words   bytes   unused memory
```
Where Do These Data Areas Go?

The compiler lays out data areas

- Where are they located at runtime?
- How does the code find the data areas?

The executable assumes that it has its own virtual address space

- Large, isolated, uninitialized tract of memory
- Compiled code manages the layout of that virtual address space
  - In collaboration with the operating system, which creates & maintains it

Virtual Address Space

| 0 | 1 | 2 | 3 | ... | n-2 | n-1 | n |
Most language runtimes layout the address space in a similar way:

- Pieces (stack, heap, code, & globals) may move, but all will be there.
- Stack and heap grow toward each other (if heap grows).
- Arrays live on one of the stacks, in the global area, or in the heap.

The picture shows one virtual address space.

- The hardware supports one virtual address space per process.
- How does a virtual address space map into physical memory?
Multiple Virtual Address Spaces?

The Big Picture

Compiler’s view

OS’ view

1980 Hardware view

virtual address spaces (one per process)

TLB is an address cache used by the OS to speed virtual-to-physical address translation. A processor may have > 1 level of TLB.