Intermediate Representations
The Glue That Holds a Compiler Together

Comp 412
Where In The Course Are We?

**Obvious answer:** *at the start of Chapter 5 in EaC2e*

**More important answer**

- We are on the cusp of the art, science, & engineering of compilation
- Scanning & parsing are applications of automata theory
- Context-sensitive analysis (**AHSDT**) is mostly software engineering
- The mid-section of the course will focus on issues where the compiler writer needs to choose among alternatives
  - The choices matter; they affect the quality of compiled code
  - There may be no “best answer” or “best practice”

**To my mind, the fun begins at this point**
Intermediate Representations

Front End emits IR for the rest of the compiler to use

- Scanner & parser work from the syntax of the source code
  - Parser emits IR using AHSDT or some similar technique
- Rest of the compiler works from IR
  - Analyzes IR to learn about the code
  - Transforms IR to improve final code
- IR determines what a compiler can do to the code
  - Can only manipulate details that are represented in the IR
Intermediate Representations

IR is the vehicle that carries information between phases

- **Front end**: produces an IR version of the input program
- **Optimizer**: transforms the IR into an equivalent IR that runs faster
  - Each “pass” reads and writes IR
- **Back end**: systematically transforms the IR into native code

IR determines both the compiler’s ambition & its chances for success

- The compiler’s knowledge of the code is encoded in the IR
- The compiler can only manipulate what is represented by the IR
Decisions in IR design affect the speed and efficiency of the compiler

Some important IR properties
- Ease of generation
- Ease of manipulation
- Cost of manipulation
- Procedure size
- Expressiveness
- Level of abstraction

Example

Ease of Manipulation: in your register allocator lab, some of you used Java’s ArrayList class. ArrayList provides many useful features that you did not need to implement yourself.

Cost of Manipulation: ArrayList is efficient, unless you are inserting items into the middle of a long list, as occurs when inserting spills and restores.

Impact on the Compiler: to combat the cost of insertion, some of you printed the output on the fly rather than updating the IR

The importance of different properties varies between compilers
⇒ Selecting an appropriate IR for a compiler is critical
Decisions in IR design affect the speed and efficiency of the compiler

Some important IR properties

- Ease of generation
- Ease of manipulation
- Cost of manipulation
- Procedure size
- Expressiveness
- Level of abstraction

Example

**Expressiveness:** a compiler with a near-source level IR may have trouble representing the results of some optimizations.

```plaintext
for i = 0 to n
for j = 0 to m
    a[i,j] = 0
    p = &a[0,0]
    for i = 0 to n * m
        *p++ = 0
```

**Simple initialization**

**After OSR**

The implementation of operator strength reduction (OSR) can only produce this result if the IR can represent *p++. (§ 10.7.2 in EaC2e.)

The importance of different properties varies between compilers

⇒ Selecting an appropriate IR for a compiler is critical
Intermediate Representations

Decisions in IR design affect the speed and efficiency of the compiler

Some important IR properties

- Ease of generation
- Ease of manipulation
- Cost of manipulation
- Procedure size
- Expressiveness
- Level of abstraction

Example

Level of Abstraction: copying a string is a complex operation that involves an internal loop. Explicitly representing the loop and its details exposes all those details to uniform optimization (a good thing).

Explicitly representing the loop makes it difficult to move the copy to another location in the code — moving control flow constructs is difficult, at best.

Representing the copy as a single operation, like the S370 mvcl makes it easy to move.

The importance of different properties varies between compilers

⇒ Selecting an appropriate IR for a compiler is critical
Intermediate Representations

Today, we will focus on representing the operations in the code

- Arithmetic expressions
- Assignment
- Control-flow in the program

Later, we will come back and talk about representing names & objects

- Symbol tables
- Renaming
- Storage models
Taxonomy of Intermediate Representations

Three major categories

• Structural IRs
  – Graphically oriented
  – Heavily used in source-to-source translators
  – Tend to be large

• Linear IRs
  – Pseudo-code for an abstract machine
  – Level of abstraction varies
  – Simple, compact data structures
  – Easier to rearrange

• Hybrid IRs
  – Combination of graphs and linear code
  – Example: control-flow graph

Examples:
- Trees, DAGs
- 3 address code
- Stack machine code
- Control-flow graph
- SSA Form
Level of Abstraction

The level of detail exposed in an IR influences the profitability and feasibility of different optimizations.

Here are two different representations of an array reference:

```
loadI 1  => r_1
sub   r_j, r_1 => r_2
loadI 10 => r_3
mult  r_2, r_3 => r_4
sub   r_i, r_1 => r_5
add   r_4, r_5 => r_6
loadI @A => r_7
add   r_7, r_6 => r_8
load  r_8 => r_{Aij}
```

High-level Abstract Syntax Tree
Good for memory disambiguation

Low-level Linear Code
Good for optimizing the address calculation
Level of Abstraction

People tend to confuse level of abstraction with structure

- Structural IRs are often considered high-level
- Linear IRs are often considered low-level
- Those generalizations are not necessarily true

![Low level AST](image)

Low level AST

load

loadArray $A, i, j$

High-level linear code

In Chapter 11 of EaC2e, we will see trees that have a lower level of abstraction than the machine code
A syntax tree represents the front end’s parse of the code, in detail.

Syntax trees are often used in source-to-source systems:
- Captures the precise (syntactic) form of the input program
- Has all of the detail that you could need
  - Compiler generated all the detail it has from the parse

Syntax trees tend to be inefficient:
- Lots of unnecessary nodes and edges
- Lots of implicit detail that might be useful to represent explicitly

Syntax trees can be represented with a linear notation (e.g., prefix or postfix)

Parse tree for \(x - 2 \times y\)
An abstract syntax tree is the procedure’s parse tree with the nodes for most non-terminal nodes removed

\[ \text{AST for } x - 2 * y \]

- **ASTs** are space efficient trees that capture most of the interesting information found in a syntax tree
  - Can regenerate source code in a treewalk, with a little cleverness
  - Many fewer nodes and edges than in a syntax tree.
- **S-expressions** in Scheme or Lisp, are (essentially) **ASTs**

In practice, **ASTs** tend to be large — not because they must, but they grow (see the digression on page 228 in EaC2e).
A directed acyclic graph (DAG) is an AST with a unique node for each value (an AST with sharing)

- Makes sharing explicit
- Encodes redundancy

If the compiler uses graphical IRs, a DAG is a natural way to represent redundancy.

With two copies of the same expression, the compiler may be able to arrange the code to evaluate it only once.

An easy way to generate DAG-like structures is to hash the values & store weak references.
See Wikipedia article.
See also “value numbering” in §8.4.1 of EaC2e.
Implementing Trees

In earlier courses, you learned to build purpose-built trees (or to use some generic library)

- Nodes connected by edges
- Nodes of various types and arities
- Allocating nodes of different sizes complicates both allocation & fragmentation in the heap (malloc())

Knuth showed that you can map any arbitrary tree onto a binary tree

- Allocate uniform-size, binary nodes
- Two “pointers”: child and sibling
- Simplify allocation and traversal

1 See Knuth Volume 1, pp. 332-334; also EaC2e pp. 744-746
Dependence Graph

Compilers use a dependence graph to understand & preserve the flow of values in a block

1. `loadI 8 => r1`
2. `loadI 12 => r2`
3. `mult r1, r2 => r3`
4. `add r1,r3 => r4`

**Original Code**

**Dependence Graph**

- Typically built as a secondary IR for scheduling or optimization
- Exposes the constraints on execution order due to flow of value
- Critical for detection of parallelism, for reordering iteration spaces of loops (memory hierarchy optimization), & for scheduling

**In Lab 3, you will build dependence graphs.**
Call Graph

When compilers try to optimize multiple procedures together, they build a call graph to represent the flow of control between procedures

- Nodes in the call graph represent procedures
- Edges represent individual calls, making it a multi-graph
  - Each edge represents a different “calling environment”

Call graphs are arbitrary graphs

- Recursion (direct or indirect) creates cycles
- Complex call patterns create complex graphs
  - Think about a recursive descent parser
- Some interprocedural optimizations change the call graph as they manipulate it
  - Think about inline substitution or procedure cloning
The role of string data

Principle:

*For the sake of compactness and efficiency, the compiler should almost never store a string value in the IR*

- Strings are expensive to compare
- Strings are large relative to their information content
- In most cases, we need just one copy of a string

How to handle strings in your IR

- Use a hash table or map to convert each string to a small integer
- Represent the string with the integer
- Compare strings as integers
Taxonomy of Intermediate Representations

Three major categories

• Structural IRs
  – Graphically oriented
  – Heavily used in source-to-source translators
  – Tend to be large

• Linear IRs
  – Pseudo-code for an abstract machine
  – Level of abstraction varies
  – Simple, compact data structures
  – Easier to rearrange

• Hybrid IRs
  – Combination of graphs and linear code
  – Example: control-flow graph

Examples:
- Trees, DAGs
- Example:
  - 3 address code
  - Stack machine code
- Example:
  - Control-flow graph
  - SSA Form
Three Address Code

Several different representations of three address code
• In general, three address code has statements of the form:
  \[ x \leftarrow y \ op \ z \]
  With 1 operator (\(\ op\)) and, at most, 3 names (\(x, y, \& z\))

Example:
\[ z \leftarrow x - 2 \times y \]
becomes

Advantages:
• Resembles many real machines
• Introduces a new set of names
• Compact form

See Lab 1 and Lab 3

The concept of “three address code” has many implementations.
Three Address Code: As Quadruples

Naïve representation of three address code

- Table of \( k \times 4 \) small integers
- Simple record structure
- Easy, albeit slow, to reorder
- Explicit names

\[
\begin{align*}
\text{load} & \quad \text{r1, y} \\
\text{loadI} & \quad \text{r2, 2} \\
\text{mult} & \quad \text{r3, r2, r1} \\
\text{load} & \quad \text{r4, x} \\
\text{sub} & \quad \text{r5, r4, r3}
\end{align*}
\]

RISC assembly code

Quadruples

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Op₁</th>
<th>Op₂</th>
<th>Op₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>1</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>loadI</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>mult</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>load</td>
<td>4</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

Store opcode & operands as small integers, of course.
Three Address Code: As Triples

- Index used as implicit name
- 25% less space consumed than quads
- Much harder to reorder

<table>
<thead>
<tr>
<th>Implicit Name</th>
<th>Opcode</th>
<th>Op₁</th>
<th>Op₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>load</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>(2)</td>
<td>loadI</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>(3)</td>
<td>mult</td>
<td>(1)</td>
<td>(2)</td>
</tr>
<tr>
<td>(4)</td>
<td>load</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>(5)</td>
<td>sub</td>
<td>(4)</td>
<td>(3)</td>
</tr>
</tbody>
</table>

Implicit names occupy no space

If you squint and look at this sideways, it looks like a tree, with all the pointers ...

Remember, for a long time, 640 KB was a lot of RAM
Three Address Code: As Indirect Triples

- List first triple in each statement
- Implicit name space for statements
- Uses more space than triples, but easier to reorder

<table>
<thead>
<tr>
<th>Stmt List</th>
<th>Implicit Names</th>
<th>Indirect Triples</th>
</tr>
</thead>
<tbody>
<tr>
<td>(100)</td>
<td>(100)</td>
<td>load y</td>
</tr>
<tr>
<td>(105)</td>
<td>(101)</td>
<td>loadI 2</td>
</tr>
<tr>
<td></td>
<td>(102)</td>
<td>mult (100) (101)</td>
</tr>
<tr>
<td>(103)</td>
<td></td>
<td>load x</td>
</tr>
<tr>
<td>(104)</td>
<td></td>
<td>sub (103) (102)</td>
</tr>
</tbody>
</table>

**Standard trick:** when you need the ability to rearrange objects in memory, add a level of indirection.

- Major tradeoff between quads and triples is compactness versus ease of manipulation
  - In the past compile-time space was critical
  - Today, speed may be more important
Original Sources

FIGURE 11.1. An ALGOL Program Segment.

(1) BLOCK
(2) - I, J
(3) BOUNDS 1, (2)
(4) ADEC A
(5) := 0, K
(6) - I, J
(7) BMZ (13), (6)
(8) - I, J
(9) * A[ (8) ], 6

(10) + K, (9)
(11) := (10), K
(12) BR (18)
(13) + I, 1
(14) := (13), I
(15) + I, 1
(16) := (15), I
(17) BRL L
(18) BLCKEND

FIGURE 11.7. Triples for Program Segment of Figure 11.1.

**Linear IRs**

Original Sources

BEGIN INTEGER K;
ARRAY A[1:I-J];
K := 0;
L: IF I > J
   THEN K := K + A[I-J]*6
   ELSE BEGIN I := I+1; I := I+1; GO TO L END
END

**FIGURE 11.1.** An ALGOL Program Segment.

1. (1) 10. (8)
2. (2) 11. (9)
3. (3) 12. (10)
4. (4) 13. (11)
5. (5) 14. (12)
6. (6) 15. (11)
7. (6) 16. (12)
8. (2) 17. (13)
9. (7) 18. (14)

**FIGURE 11.8.** Indirect Triples for Figure 11.1.

Two-Address Code

Two-address code allows statements of the form

\[ x \leftarrow x \, op \, y \]

Each operation has 1 operator (\(op\)) and, at most, 2 names (\(x\) and \(y\))

Example

\[ z \leftarrow x - 2 \times y \quad becomes \]

\[ \begin{align*}
  t_1 & \leftarrow 2 \\
  t_2 & \leftarrow \text{load } y \\
  t_2 & \leftarrow t_2 \times t_1 \\
  z & \leftarrow \text{load } x \\
  z & \leftarrow z - t_2
\end{align*} \]

We write:

\[ r_1 + r_2 \Rightarrow r_2 \]

as:

\[ \text{add } r_1, r_2 \]

Problems

• Difficult name space
  – Destructive operations make reuse hard
  – Good model for machines with destructive ops (PDP-11, x86)

• We would like destructive operations to become a thing of the past

Not many arguments in favor of two-address code
Stack Machine Code

Originally used for stack-based computers, now Java

- Example:
  \[ x - 2 \times y \]
  becomes
  - `push x`
  - `push 2`
  - `push y`
  - `multiply`
  - `subtract`

Advantages

- Compact form
- Introduced names are *implicit*, not *explicit*
- Simple to generate and execute code

Useful where code is transmitted over slow communication links or where memory is limited

- Java bytecode was designed for transmission over slow links
- Follows a long line of bytecode-like IRs designed to be compact

In a stack machine, most operations are destructive.
Taxonomy of Intermediate Representations

Three major categories

• Structural IRs
  – Graphically oriented
  – Heavily used in source-to-source translators
  – Tend to be large
  Examples: Trees, DAGs

• Linear IRs
  – Pseudo-code for an abstract machine
  – Level of abstraction varies
  – Simple, compact data structures
  – Easier to rearrange
  Examples: 3 address code Stack machine code

• Hybrid IRs
  – Combination of graphs and linear code
  – Example: control-flow graph
  Examples: Control-flow graph SSA Form
Control-flow Graph

Models the transfer of control in the procedure

- Nodes in the graph are basic blocks
  - Can be represented with quads or any other linear representation
- Edges in the graph represent control flow

Example

```
if (x = y)
  a ← 2
  b ← 5
  c ← a * b
  a ← 3
  b ← 4
```

Basic blocks: Maximal length sequences of straightline code

Edges represent the branches & jumps at the ends of blocks.

Implementations:
See Figures B.3 and B.4 in Appendix B of EaC2e
The Fundamental Idea: A Name Space In The IR That Simplifies Analysis

1. Each name in the code is defined exactly once
2. Each use refers to exactly one name

Why do we want these properties?

• Provide a unique name for each value
  — Simplifies many optimizations by eliminating kills  
  \textit{(def no longer kills name)}
• Expose both the flow of values & the ranges of values
  — Can simplify implementation of both analyses & transformations

Why not use SSA for everything?

• Some compilers, in effect, do use SSA as their primary IR
  — LLVM/CLANG, FLANG
• Some aspects of SSA are not easily implementable in code
  — \( \phi \)-functions have no direct analog in most ISAs
Static Single-Assignment Form

The Fundamental Idea: A Name Space In The IR That Simplifies Analysis

1. Each name in the code is defined exactly once
2. Each use refers to exactly one name

What is easy?

• Straight-line code is trivial
  – New name at every definition
  – Add a subscript, bump it at definition, rewrite with current subscript

• Splits in the CFG are trivial
  – Each block inherits the name space of its (sole) predecessor

What is hard?

• Joins (or merge points) in the CFG are hard
  – Block can inherit two names for the same original program variable
  – Need a mechanism to reconcile such conflicts
Static Single Assignment Form

Reconciling the two rules of SSA form

- Introduce $\phi$-functions at join points to create new, merged values
  - $\phi$-functions are glorified parallel copies
  - $\phi$-function takes one argument per incoming CFG edge
  - $\phi$-function selects argument that corresponds to control flow

\[
\begin{align*}
  x & \leftarrow \ldots \\
  y & \leftarrow \ldots \\
  \text{while } (x < k) \\
  & \quad \begin{aligned}
  x & \leftarrow x + 1 \\
  y & \leftarrow y + x
  \end{aligned}
\end{align*}
\]

Original Code

\[
\begin{align*}
  x_0 & \leftarrow \ldots \\
  y_0 & \leftarrow \ldots \\
  \text{while } (x < k) \\
  & \quad \begin{aligned}
  x_1 & \leftarrow \phi(x_0, x_2) \\
  y_1 & \leftarrow \phi(y_0, y_2) \\
  x_2 & \leftarrow x_1 + 1 \\
  y_2 & \leftarrow y_1 + x_2
  \end{aligned}
\end{align*}
\]

Same Code in SSA Form

Strengths of SSA-form

- Sharper analysis
- Faster, cleaner algorithms (sometimes)
Combination of IRs

Many compilers use a CFG to represent control flow and a linear IR to represent code in blocks

• This hybrid IR has the advantages of a graph
  – Easy navigation between blocks

• And the advantages of a linear IR
  – Explicit, low-level detail & operation sequence

Strengths

• Good for understanding control-flow issues

• Good for understanding flow of data (program analysis)

In lab 3, you will build and use a dependence graph, linked to a basic block. The dependence graph records the flow of values, while the block records both the details of those operations & their relative order.
Using Multiple Representations

• Repeatedly lower the level of abstraction in the IR
  — Each IR is suited towards certain optimizations

• Example: the Open64 compiler
  — WHIRL intermediate format
    → *Consists of 5 different IRs that are progressively more detailed and less abstract*
    → *Each successive IR focuses on a different set of challenges & opportunities*
  — Translation is a monotonic lowering of level of abstraction
    → *Compilers are good at lowering level of abstraction & not so good at raising it*
Memory Models

An IR usually incorporates a memory model, explicit or implicit.

- **Register-to-register model**
  - Keep all values that can legally be stored in a register in registers
  - Ignore machine limitations on number of registers
  - Compiler back-end must insert loads and stores

- **Memory-to-memory model**
  - Keep all values in memory
  - Only promote values to registers directly before they are used
  - Compiler back-end can remove loads and stores

**Compilers for RISC machines usually use a register-to-register model**

- Closely reflects RISC instruction sets
- Register use is explicit and well-modelled
The Rest of the Story...

Representing the code is only part of an IR

Other components are necessary

• Symbol table
  – Every name in the program

• Constant table
  – Representation, type
  – Storage class, offset

• Storage map
  – Overall storage layout
  – Overlap information
  – Virtual register assignments

We will return to these structures, their motivation, and their use over the next several lectures.
A parser’s other activities can be termed “semantic elaboration.”

- It may perform many tasks, including
  - Type checking
  - Code generation
  - Storage layout
  - Error checking

- It might create an IR & interpret that IR to produce answers

- It might pretty-print the code back to a file in some modified form.

- Semantic elaboration makes parsers quite versatile tools.
Example of **SDT**

Consider a simple example: building an AST in a recursive-descent parser

$$
\text{Stmts} \rightarrow \text{Assign \ StmtList} \\
\text{StmtList} \rightarrow \text{Assign \ StmtList} | \epsilon \\
\text{Assign} \rightarrow \text{LHS} \leftarrow \text{RHS}
$$

Simple little grammar (similar to a *ProductionList*)

```cpp
bool Stmts() {
    if ( Assign() )
        then return StmtList()
    else
        print "Looking for assignment, found none"
    return false;
}
```

Duplicate **RHS** for **Stmts** & **StmtList** ensures that **Stmts** contains at least one **Assign**
Example of SDT

Consider a simple example: building an AST in a recursive-descent parser

\[
\begin{array}{ll}
\text{Stmts} & \rightarrow \text{Assign} \ \text{StmtList} \\
\text{StmtList} & \rightarrow \text{Assign} \ \text{StmtList} \\
& \ | \ \text{epsilon} \\
\text{Assign} & \rightarrow \text{LHS} \leftarrow \text{RHS}
\end{array}
\]

Simple little grammar (similar to a ProductionList)

The Plan is easy:

1. Make Stmts() return a (bool,tree) pair (and Assign, and StmtList, ...)
2. Build the appropriate tree structure before the “returns”

```
bool Stmts() {
    if ( Assign() )
        then return StmtList()
    else
        print “Looking for assignment, found none”
        return false;
}
```

Duplicate RHS for Stmts & StmtList ensures that Stmts contains at least one Assign
Example of SDT

Building the IR in Stmts() makes matters somewhat more complex

```c
pair(bool,tree) Stmts() {
    result1 = Assign()
    if (result1.bool) then {
        result2 = StmtList()
        if (result2.bool) then {
            if (result2.tree == empty) then return result1;
            else return (true, cons(result1,tree,result2.tree))
        } else 
            print "Looking for assignment, found none"
        return (false, empty)
    } else
    return (false, empty)
}
```

The Plan is easy:

1. Make Stmts(), Assign(), StmtList(), ...
2. Build the appropriate tree structure before the “returns”

The Implementation is a little messy

Duplicate RHS for Stmts & StmtList ensures that Stmts contains at least one Assign
What About AHSDT In an LR(1) Parser?

We specify SDT actions using a simple notation

1. `Number → DigitList { $$ = $1; }`
2. `DigitList → DigitList digit { $$ = $1 * 10 + $2; }`
3. `| digit { $$ = $2; }`

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
Fitting **AHSDT** into the **LR(1)** Skeleton Parser

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()`

```
stack.push(INVALID);
stack.push(s0);  // initial state
word = scanner.next_word();
loop forever {
    s = stack.top();
    if ( ACTION[s,word] == "reduce $A \rightarrow \beta" ) then {
        stack.popnum(2*|\beta|);  // pop 2*|\beta| symbols
        s = stack.top();
        stack.push(A);  // push LHS, A
        stack.push(GOTO[s,A]);  // push next state
    }
    else if ( ACTION[s,word] == "shift $s_i" ) 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 \textbf{AHSDT} into the \textbf{LR(1)} Skeleton Parser

```
stack.push(INVALID);
stack.push(s_0); // initial state
word = scanner.next_word();
loop forever {
    s = stack.top();
    if ( ACTION[s,word] == "$reduce\ A \rightarrow \beta\$" ) then {
        stack.popnum(2*|\beta|); // pop 2*|\beta| symbols
        s = stack.top();
        stack.push(A); // push LHS, A
        stack.push(GOTO[s,A]); // push next state
    }
    else if ( ACTION[s,word] == "$shift\ s_i\$" ) then {
        stack.push(word); stack.push(s_i);
        word \leftarrow scanner.next_word();
    }
    else if ( ACTION[s,word] == "$accept\" & word == EOF )
        then break;
    else throw a syntax error;
}
report success;
```

**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
  - \textit{Work()} takes the stack as input (conceptually) and returns the value for the reduction it processes
  - \textit{Shift} creates initial values
Fitting AHSĐT 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

```c
{ 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

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

How do we implement \textit{Work()}?

- \textit{Work()} takes 2 arguments: the stack and a production number
- \textit{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, $3 \ldots$ macros translate into references into the stack
  - The $$ macro translates into the return value

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

$$ \text{translates to } r$$

\$i \text{ translates to the stack location } 3 \ast (|\beta| - i + 1) \text{ units down from stacktop}

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