Abstract Syntax Tree

Related terms:

Directed Acyclic Graph, Compiler, Control-Flow Graph, Refactoring, Sharding, Parse Tree, Clone Detection, Bitcoin

View all Topics

Modernization Standards Roadmap

William Ulrich, in Information Systems Transformation, 2010

Abstract syntax tree metamodel

This ASTM was established to represent software at a very granular level of procedural logic, data definition, and workflow composition. ASTM can provide this granular level of information to KDM to augment the KDM view of a system. As a standard, ASTM can stand alone and supports tools geared at the complete, functionally equivalent refactoring and transformation of a system from one platform and language environment to a target platform and language environment.

Some background on the concept of the abstract syntax tree (AST) helps clarify the purpose of the ASTM. ASTs are models of software that represent software artifacts using data structures that represent the types of language constructs, their compositional relationships to other language constructs, and a set of direct and derived properties associated with each language construct. The AST is derived by analyzing software artifacts and provides a way to create a representation of those software artifacts.

An AST is an extensible, formal representation of the syntactical structure of software that is amenable to formal analysis techniques. It is possible to traverse the AST and reconstruct the "surface syntax" of the system or reconstitute it in textual form from the abstract structures. While the use of AST structures for the abstract representation of the structure of software has become an accepted practice for modeling software, the format of AST structures and the mechanisms for representation and interchange of AST models were not standardized prior to the formalization of the ASTM.

AST interchange, via the ASTM, facilitates exchanging software models in standard formats among tools. The ability to freely exchange software models between tools provides organizations with the ability to use advanced model-driven tools for software analysis and modernization. The ASTM has these attributes:

- ASTM is language and platform independent, but can be extended as needed.
- ASTM uses XMI formats for tool-based metadata exchange.
- Generic Abstract Syntax Tree Metamodel (GASTM) represents a generic set of language modeling elements common across numerous languages. Language Specific Abstract Syntax Tree Metamodel (SASTM) represents particular languages such as Ada, C, FORTRAN, and Java.
- Proprietary Abstract Syntax Tree Metamodel (PASTM) expresses ASTs for languages such as Ada, C, COBOL, etc., modeled in formats inconsistent with MOF, the GSATM, or SASTM.

Figure 3.3 represents the structure of the ASTM, including the SASTM, GASTM, and PASTM.

Figure 3.3. ASTM modeling framework.[7]

The ASTM standard, which was officially adopted as a standard in 2009, will continue to evolve as it expands to support more languages and environments. It will also support the evolution of other ADM standards as they change and mature.

> Read full chapter

Compilers

Keith D. Cooper, ... Linda Torczon, in Encyclopedia of Physical Science and Technology (Third Edition), 2003

III Internal Representations

Once the compiler is broken into distinct phases, it needs an internal representation to transmit the program between them. This internal form becomes the definitive representation of the program—the compiler does not retain the original source program. Compilers use a variety of internal forms. The selection of a particular internal form is one of the critical design decisions that a compiler writer must make. Internal forms capture different properties of the program; thus, different forms are appropriate for different tasks. The two most common internal representations—the abstract syntax tree and three-address code—mimic the form of the program at different points in translation.

- The abstract syntax tree (AST) resembles the parse tree for the input program. It includes the important syntactic structure of the program while omitting any nonterminals that are not needed to understand that structure. Because of its ties to the source-language syntax, an AST retains concise representations for most of the abstractions in the source language. This makes it the IR of choice for analyses and transformations that are tied to source program structure, such as the high-level transformations discussed in Section VC.
- *Three-address code* resembles the assembly code of a typical microprocessor. It consists of a sequence of operations with an implicit order. Each operation has an operator, one or two input arguments, and a destination argument. A typical three-address code represents some of the relevant features of the target machine, including a realistic memory model, branches and labels for changing the flow of control, and a specified evaluation order for all the expressions. Because programs expressed in three-address code must provide an explicit implementation for all of the source language's abstractions, this kind of IR is well suited to analyses and transformations that attack the overhead of implementing those abstractions.

To see the difference between an AST and a three-address code, consider representing an assignment statement **a**[**i**] ← **b** * **c** in each. Assume that **a** is a vector of 100 elements (numbered 0–99) and that **b** and **c** are scalars.

The AST for the assignment, shown on the left, captures the essence of the source-language statement. It is easy to see how a simple in-order treewalk could reprint the original assignment statement. However, it shows none of the details about how the assignment can be implemented. The three-address code for the assignment, shown on the right, loses any obvious connection to the source-language statement. It imposes an evaluation order on the statement: first **b**, then **c**, then **b * c**, then **i**, then **a[i]**, and, finally, the assignment. It uses the notation @**b** to refer to **b**'s address in memory—a concept missing completely from the AST.

Many compilers use more than one IR. These compilers shift between representations so that they can use the most appropriate form in each stage of translation.

> Read full chapter

27th European Symposium on Computer Aided Process Engineering

Arne Tobias Elve, Heinz A. Preisig, in Computer Aided Chemical Engineering, 2017

3.3 Making executable equations

Having established the abstract syntax tree for each variable and the index sets it is possible to translate the expression into a targeted language by applying templates for the operators. A parser interprets a string expression captured in the ontology and translates it to executable code in the targeted language. The string expression is formulated using eight operators that manipulate the index set and eight unitary functions that do element-wise operations without changing the structure of the variable. These operators are implemented in Python, using the numerical library Numpy, (see table 1). In addition, there are eight unitary functions that map element by element for any indexed variable. The implemented unitary functions are; sin, cos, sign, sqrt, log10, ln, inv and abs. These equation templates can be nested to form executable expressions. For the example in fig. 3, the Python version of this expression is generated as:

```
mu.ex = add(mu_Orr, (khatriRaoProduct ((expandproduct (Rg, T)), (ln (fn)))))
```

The variables mu, mu_Orr, Rg, T and fn are implemented as variable containing units, index sets, executable expressions and values that have been updated using the respective executable expressions before the chemical potential is calculated. The sequence in which the variables are calculated is determined by traversing the abstract syntax tree.

> Read full chapter

Anti-Pattern Detection

Fabio Palomba, ... Rocco Oliveto, in Advances in Computers, 2014

2.3.3 Analysis of the Detection Accuracy

The detection methods based on AST [19,20] have been applied on one process-control system written in C, having 400 KLOC. The authors found that most of detected clones were of small size and they were functions performing the same operations in different places. The main problem of this approach is that it is able to determine only exact tree match. The empirical evaluation conducted on DECKARD involved large systems as JDK and the Linux kernel and showed that the tool performs significantly better than the other state-of-art technique to detect clones [21].

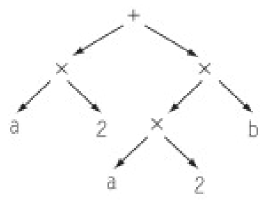
> Read full chapter

Intermediate Representations

Keith D. Cooper, Linda Torczon, in Engineering a Compiler (Second Edition), 2012

Abstract Syntax Trees

The abstract syntax tree (ast) retains the essential structure of the parse tree but eliminates the extraneous nodes. The precedence and meaning of the expression remain, but extraneous nodes have disappeared. Here is the ast for a × 2 + a × 2 × b:



Abstract Syntax Tree

An ast is a contraction of the parse tree that omits most nodes for nonterminal symbols.

The ast is a near-source-level representation. Because of its rough correspondence to a parse tree, the parser can built an ast directly (see Section 4.4.2).

asts have been used in many practical compiler systems. Source-to-source systems, including syntax-directed editors and automatic parallelization tools, often use an ast from which source code can easily be regenerated. The S-expressions found in Lisp and Scheme implementations are, essentially, asts.

Even when the ast is used as a near-source-level representation, representation choices affect usability. For example, the ast in the □*n* Programming Environment used the subtree shown in the margin to represent a complex constant in fortran, written (c_1, c_2). This choice worked well for the syntax-directed editor, in which the programmer was able to change c_1 and c_2 independently; the pair node corresponded to the parentheses and the comma.

This pair format, however, proved problematic for the compiler. Each part of the compiler that dealt with constants needed special-case code for complex constants. All other constants were represented with a single node that contained a pointer to the constant's actual text. Using a similar format for complex constants would have complicated some operations, such as editing the complex constants and loading them into registers. It would have simplified others, such as comparing two constants. Taken over the entire system, the simplifications would likely have outweighed the complications.

Storage Efficiency and Graphical Representations

Many practical systems have used abstract syntax trees to represent the source text being translated. A common problem encountered in these systems is the size of the ast relative to the input text. Large data structures can limit the size of programs that the tools can handle.

The ast nodes in the □*n* Programming Environment were large enough that they posed a problem for the limited memory systems of 1980s workstations. The cost of disk i/o for the trees slowed down all the □*n* tools.

No single problem leads to this explosion in ast size. □*n* had only one kind of node, so that structure included all the fields needed by any node. This simplified allocation but increased the node size. (Roughly half the nodes were leaves, which need no pointers to children.) In other systems, the nodes grow through the addition of myriad minor fields used by one pass or another in the compiler. Sometimes, the node size increases over time, as new features and passes are added.

Careful attention to the form and content of the ast can shrink its size. In □*n*, we built programs to analyze the contents of the ast and how the ast was used. We combined some fields and eliminated others. (In some cases, it was less expensive to recompute information than to write it and read it.) In a few cases, we used hash linking to record unusual facts—using one bit in the field that stores each node's type to indicate the presence of additional information stored in a hash table. (This scheme reduced the space devoted to fields that were rarely used.) To record the ast on disk, we converted it to a linear representation with a preorder treewalk; this eliminated the need to record any internal pointers.

In □*n*, these changes reduced the size of asts in memory by roughly 75 percent. On disk, after the pointers were removed, the files were about half the size of their memory representation. These changes let □*n* handle larger programs and made the tools more responsive.

Abstract syntax trees have found widespread use. Many compilers and interpreters use them; the level of abstraction that those systems need varies widely. If the compiler generates source code as its output, the ast typically has source-level abstractions. If the compiler generates assembly code, the final version of the ast is usually at or below the abstraction level of the machine's instruction set.

> Read full chapter

Legacy System Modernization of the Engineering Operational Sequencing System (EOSS)*

Mark Kramer, Philip H. Newcomb, in Information Systems Transformation, 2010

LSM Technology

JPGEN™ is TSRI's technology for generating parsers that process code to generate abstract syntax tree (AST) models and printers that generate code from AST models from a grammar specification for a computer language. JRGEN™ is TSRI's technology for automatically recognizing AST patterns and transforming AST models. TSRI employs JPGEN™ and JRGEN™ in a highly disciplined process that transforms the entire legacy application into an Intermediate Object Model (IOM) upon which a sequence of transformation and refactoring operations are applied before target code is generated from the AST models.

Parsing into the Legacy AST model is carried out by a parser generated by JPGEN™ and basic semantics generated by a constrainer generated by JRGEN™. The transformation between the Legacy VAX BASIC AST model and the IOM AST model is carried out by VAXBASIC2IOM transformation rules generated by JRGEN™. The transformation between the IOM and the target Java AST is carried out by the IOM2Java transformation rules generated by JRGEN™.

> Read full chapter

Towards a New Formal SDL Semantics based on Abstract State Machines

U. Glässer, ... A. Prinz, in SDL '99, 1999

2.4 ASM Representation of SDL Specifications

SDL specifications are represented in the ASM model based on their AST. In Figure 2, an excerpt of the AST for the running example is shown. This AST is then encoded by a finite collection of ASM objects from a static domain *DEFINITION* in combination with various functions defined thereon. By means of these functions, the definitions are related to each other and the relevant information can be extracted. Emphasizing the tight relation between SDL specifications and their ASM representation, the terminology and the notation of the AS are reused here. That is, function names denoting AST components in the machine model are identical to the non-terminals used in the AS definition:

Figure 2. Abstract Syntax Tree of the SDL Specification of System1

System-definition:	*DEFINITION*
System-name:	*DEFINITION* → *NAME*
Block-definition-set:	*DEFINITION* → *DEFINITION*-set
Block-definition:	*DEFINITION* → *DEFINITION*
… *etc.*	

> Read full chapter

Processing Device Arrays with C++ Metaprogramming

Jonathan M. Cohen, in GPU Computing Gems Jade Edition, 2012

32.3 Implementation

32.3.1 Building an AST in C++

To use C++ templates to represent an AST, we need a scheme for expressing a tree as a C++ type. The basic idea is to represent a tree node (corresponding to a single operation in a C++ expression tree) as a class, which I will call an Op. All Op classes must adhere to a protocol that they implement a static function exec(int i, PARM p) that returns a value for each result array position i based on some associated parameter p. The PARM type is template parameter to the Op class, and it refers to whatever additional state may be needed to implement exec. For example, if the Op represents reading from an input array, PARM would contain the array's base pointer. A generic Op will look like this:

```
template<class PARM>
struct SomeGenericOp {
__device__ static float exec(int i, const PARM &p) {
/* return some function evaluated at i with input parameter p */
}
};
```

The combination of an Op with a specific instance of its input PARM state is called an OpWithParm:

```
template<class OP, class PARM>
struct OpWithParm {
OpWithParm(const PARM &p) : parm(p) { }
PARM parm;
__device__ float exec(int i) const { return OP::exec(i, parm); }
};
```

The device function OpWithParm::exec() calls the Op's exec routine, passing it the index i and the bound PARM state. Note that the contents of the state, represented by the parm member variable, is bound at runtime, while the function to be called, represented by the OP template parameter, is bound at compile time.

The OpWithParm functor represents some device routine that will be applied to every element in an output array to generate a value. To implement our flexible array processing API, we want to implement arbitrary C++ expressions as instantiations of the OpWithParm template. This step is accomplished using a variant of Expression Templates, where C++ template type calculus can be used to trigger arbitrary code generation by the template preprocessor.

For example, say we want to trigger evaluation of the expression (i + 5) for each index i. The expression tree for this function is shown in Figure 32.1.

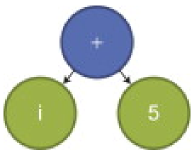


Figure 32.1. Tree representing the value i + 5.

This tree consists of two types of nodes: leaves and internal nodes. Leaf nodes, colored light gray, compute some value directly as a function of a PARM instance and an index. Internal nodes, the top circle with the + inside, aggregate result from one or more children Op nodes. The implementation of a leaf node's exec routine directly delegates to its contained PARM instance to calculate a return value. PARM classes must implement a const member function value(int i) that computes a value directly from the index and optionally from its own state. Unlike Op::exec, this value will depend on the runtime contents of the PARM structure; therefore, it is a member function.

```
template<class PARM>
struct LeafOp {
__device__ static float exec(int i, const PARM &p) {
return p.value(i);
}
};
```

The PARM class determines how to calculate the actual value. For example, consider these three basic types:

```
struct IdentityParm {
IdentityParm(){ }
__device__ float value(int i) const {
```

```
return (float)i;
}
};
struct ConstantParm {
float _value;
ConstantParm(float f) : _value(f) { }
__device__ float value(int i) const {
return _value;
}
};
struct ArrayLookupParm {
const float *_ptr;
int _shift;
ArrayLookupParm(const DeviceArray1D &array, int shift):
_ptr(array._ptr), _shift(shift) { }
__device__ float value(int i) const {
return _ptr[(i+_shift)];
}
};
```

IdentityParm returns i and does not require any internal state. ConstantParm returns a fixed constant, which is determined at runtime. ArrayLookupParm provides a way to access a DeviceArray1D by returning the associated value, optionally shifting the lookup index by a given amount (shift). A LeafOp can be bound to any of these PARM types and then bound to a particular PARM instance via an OpWithParm to create a self-contained functor.

Expression Templates can be used to build functors without requiring any of the types to be declared by the user of the API. For example, to create a functor that wraps up the ConstantParm as a functor, the API provides a host routine constant(float) that converts from a float value to a functor. Similarly with identity():

```
OpWithParm<LeafOp<ConstantParm>, ConstantParm>
constant(float value) {
```

```
return OpWithParm<LeafOp<ConstantParm>, ConstantParm >(
ConstantParm (value));
}
OpWithParm<LeafOp<IdentityParm>, IdentityParm>
identity() {
return OpWithParm<LeafOp<IdentityParm>, IdentityParm>();
}
```

To expose the ArrayLookupParm via a functor, we provide an overloaded version of the DeviceArray1D::operator[] where the parameter represents the shift amount.

```
OpWithParm<LeafOp<ArrayLookupParm>, ArrayLookupParm>
DeviceArray1D::operator[](int shift) {
return OpWithParm<LeafOp<ArrayLookupParm>,ArrayLookupParm>(
ArrayLookupParm(g, shift));
}
```

In all of these cases so far, the functor consists of a single LeafOp. This allows expression trees consisting of only a single leaf node, which in conjunction with a way to invoke the functor for each output entry is enough to express simple assignment and copy operations.

```
template <typename T>
__global__ void kernel_assign(T functor, float *result, int size)
{
int i = blockIdx.x * blockDim.x + threadIdx.x;
if (i < size) {
result[i] = functor.exec(i);
}
}
template<typename OP, typename PARM>
DeviceArray1D &
DeviceArray1D::operator=(const OpWithParm<OP,PARM> &func)
{
```

```
kernel_assign<<<(_size+255)/256, 256>>>(func, _ptr, _size);
return *this;
}
```

DeviceArray1D::operator= is overloaded to take an OpWithParm functor as the right-hand side. The assignment operator calls kernel_assign, which fills the array with the results of the functor invoked at each index. Because the assignment operator is templated on the OP and PARM types, neither the API nor the user code needs to explicitly specify the functor types. The following assignment examples demonstrate the power of this technique — complex type calculus and template expansion can be invoked by the template preprocessor, but the expanded types remain entirely hidden from the API user by further templating the assignment routines:

```
DeviceArray1D A(100), B(100);
B = constant(5.0f); // assign 5.0f to all entries in B
A = B[0]; // copy B to A, with no offset
```

In order to create arbitrary expressions, we need a way to aggregate leaf nodes via internal nodes to create complex expressions trees. Each internal node will represent a single C++ operation, which can be a unary, binary, ternary, or general n-ary function. We will illustrate how binary functions can be expressed in our system for pairwise addition of input vectors; other cases are similar.

```
template<typename LPARM, typename RPARM>
struct ParmPair {
LPARM left;
RPARM right;
ParmPair(const LPARM &l, const RPARM &r) :
left(l), right(r) { }
};
template<class LOP, class ROP, class LPARM, class RPARM>
struct PlusOp {
__device__ static float exec(
int i, const ParmPair<LPARM, RPARM> &p) {
return LOP::exec(i,p.left) + ROP::exec(i,p.right);
}
};
```

ParmPair aggregates two PARM objects into a single struct that can be passed around as a single entity, while the PlusOp::exec adds the results of two child Op::exec calls together. Using the expression templates technique for building types automatically via template functions, operator+ is overridden to emit the appropriately instantiated PlusOp template:

```
template<class LOP, class LPARM, class ROP, class RPARM>
OpWithParm<PlusOp<LOP, LPARM, ROP, RPARM>, ParmPair<LPARM, RPARM> >
operator+(
const OpWithParm<LOP, LPARM> &left,
const OpWithParm<ROP, RPARM> &right)
{
return OpWithParm<PlusOp<LOP, ROP, LPARM, RPARM>,
ParmPair<LPARM, RPARM> >(
ParmPair<LPARM, RPARM>(left.parm, right.parm))
}
```

All C++ operators can be wrapped this way, which allows ASTs to be constructed to represent arbitrary C++ expressions. Because operator precedence is unaffected by operator overloading, the C++ parser will properly apply grouping to build an AST following standard order of operations. For example, the following listing produces the expression tree shown in Figure 32.2 and evaluates it for each entry A[i]. Leaf nodes are the three circles with the 0.5, B[i], and C[i] inside them, and internal nodes are the two circles with the * and + inside them.

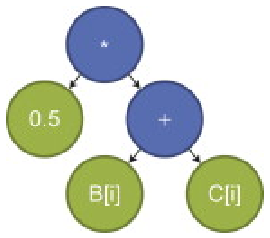


Figure 32.2. Abstract syntax tree for the expression: A = constant(0.5f) * (B[0] + C[0]).

```
DeviceArray1D A(100), B(100), C(100);
A = constant(0.5f) * (B[0] + C[0]);
```

When you compile with the “nvcc -keep” option, the output of the C++ preprocessor is written to a file that can be examined. The generated code for this expression is shown at the end of this section; it has been slightly edited for clarity. Notice that the code specifies how to walk the expression tree with sequences of .left and .right to select values from the PARM structure. Because structures are laid out at compile time, these expressions are evaluated statically and compiled into a hard-coded offset from the base of the generated_OpWithParm structure.

```
__global__ void generated_kernel_assign(
generated_OpWithParm ftor,
float *dst,
int nx)
{
int i = blockIdx.x * blockDim.x + threadIdx.x;
if (i < nx) {
dst[i] = ftor.parm.left._value *
(
ftor.parm.right.left._ptr[i+functor.parm.right.left._shift] +
ftor.parm.right.right._ptr[i+functor.parm.right.right._shift]
);
}
```

32.3.2 Handling Boundary Conditions

Finite-difference stencils require some way of handling the case when the stencil overhangs the edge of the array, known as a boundary condition. Because our framework allows for shifted reads, we need to handle what happens when we read from out-of-bounds values. Periodic boundary conditions are a simple example, in which case out-of-bounds reads “wrap around.” For example, a read from the location –1 actually returns the value stored in location _size–1.

The simplest strategy for handling out-of-bounds reads is to check all accesses in the Array LookupParm. For this, we can write a separate version called ArrayLookup-PeriodicParm:

```
struct ArrayLookupPeriodicParm {
const float *_ptr;
int _shift;
int _size;
ArrayLookupPeriodicParm(const DeviceArray1D &array, int shift) :
_ptr(array._ptr), _shift(shift), _size(array._size) { }
__device__ float value(int i) const {
return _ptr[((i+_shift)+_size)%_size];
}
};
```

Different versions of ArrayLookupParm can each implement a different policy for how to handle boundary conditions. Rather than overloading operator[], the API can provide member functions that instantiate different boundary condition handling routines such as read_periodic, read_neumann, read_dirichelet, and so on.

A downside to explicitly handling out-of-bounds accesses is that it inserts boundary handling logic before every read, even though most reads will be in-bounds. An alternate approach is to use so-called ghost cells (sometimes referred to as boundary cells, apron cells, or halos). Ghost cells are suitable for situations where out-of-bounds accesses are known to be only slightly outside the allocated range. The array is padded in both directions by a small amount, and these padded "ghost" cells can be filled in with values to achieve the desired boundary conditions. No special handling is needed for out-of-bounds reads, although an extra routine is required to fill in the ghost values appropriately.

32.3.3 Automatic Bounds Checking

Using ghost cells, we can read past the end of an array by shifting beyond the extent of the ghost cells, resulting in undefined values or memory errors. However, because we have an explicit representation for the memory access patterns, we can automatically generate a routine to validate all accesses using range calculus before the kernel is launched.

First, define a basic range class:

```
struct Range {
int _min_i, _max_i;
Range(int min_i, int max_i) : _min_i(min_i), _max_i(max_i) { }
bool contains(const Range &r) const {
return r._min_i >= _min_i && r._max_i <= _max_i;
}
Range shift(int shift) const {
return Range(_min_i + shift, _max_i + shift);
}
};
```

We also need to extend the DeviceArray1D class to support ghost cells:

```
struct DeviceArray1D {
int _padding;
float *_base_ptr;
...
DeviceArray1D(int size, int padding) : ... {
// allocate size + 2 * padding entries
cudaMalloc((void **)&_base_ptr, ...);
_ptr = _base_ptr + _padding;
...
}
~DeviceArray1D() { if (_ptr) { cudaFree(_base_ptr); } }
};
```

Given an instance of DeviceArray1D, we can obtain its range as Range(−_padding, _size+_padding−1). We also add a routine to OpWithParm that checks whether the output range is entirely inside the valid region of this functor. Note that this routine will be called from the host.

```
template<class OP, class PARM>
struct OpWithParm {
...
bool validate(const Range &rng) const {
return OP::validate(rng, parm);
}
};
```

Before the CUDA kernel gets launched, we check that the functor is valid over the desired output range.

```
DeviceArray1D::operator=(const OpWithParm<OP,PARM> &func)
{
if (!func.validate(range())) {
// run-time error
}
…
}
```

All Op classes must follow the protocol that OP::validate(Range) returns true if calls to that instance's exec method will produce valid results inside the given range, and false otherwise. Each PARM class implements the same protocol, and LeafOp::validate delegates to its PARM instance. For example, the ConstantParm::validate will always return true, while ReadArrayParm::validate will check the range against its input:

```
struct ArrayLookupParm {
…
Range _rng;
ArrayLookupParm(const DeviceArray1D &grid, int shift) :
_rng(grid.range()), … { }
bool validate(const Range &rng) const {
return _rng.contains(rng.shift(_shift));
}
};
```

Validation of a range by child nodes is recursively aggregated up the tree to the root by internal nodes. For example, PlusOp::validate checks whether both its left and right children are valid over the given range:

```
template<class LOP, class ROP, class LPARM, class RPARM>
struct PlusOp {
…
static bool validate(
const Range &rng, const ParmPair<LPARM, RPARM> &p) {

return LOP::validate(rng, p.left) && ROP::validate(rng, p.right);

}

};
```

A call to validate on the top-level Op will walk the entire AST, aggregating results up from the leaves and returning true if the entire tree is valid at the root and false otherwise. Note that this mechanism for delegating work down to the leaves and then aggregating results back up to the top is basically the same scheme we previously used to generate the CUDA kernel. In both cases, the C++ template preprocessor is used to write a function on the fly based on a traversal of the AST, dispatching to specific code-generation snippets by type. This technique could be used in a number of other ways as well, such as to generate a multicore version of the CUDA routines or to count how many floating-point operations will be executed by the CUDA kernel.

> Read full chapter

Veterans Health Administration's VistA MUMPS Modernization Pilot*

Philip H. Newcomb, Robert Couch, in Information Systems Transformation, 2010

IOM to Java Transformation

Transformation mapping consists of rewrite rules that are applied to the AST constructs that result from the application of a parser to the sentences of a language. A collection of transformation mapping rules is written that define a set of model-to-model mapping from the original legacy language into the target language. The rule below handles the mapping from the IOM into Java. Much more complex rules than the one shown next are required for transforming more complex language features.

// IO::ADD-EXPRESSION to java::ADD Transformation MappingJform (< **IOM::ADD-EXPRESSION** Omitted: TSRI proprietary >) ==> < **java::ADD** *Omitted: TSRI proprietary* > ;

> Read full chapter

Instruction Selection

Keith D. Cooper, Linda Torczon, in Engineering a Compiler (Second Edition), 2012

11.4.1 Rewrite Rules

The compiler writer encodes the relationships between operation trees and subtrees in the ast as a set of *rewrite rules*. The rule set includes one or more rules for every kind of node in the ast. A rewrite rule consists of a production in a tree grammar, a code template, and an associated cost. Figure 11.4 shows a set of rewrite rules for tiling our low-level ast with iloc operations.

	Production			**Cost**	**Code Template**			
1	Goal	→	Assign	0				
2	Assign	→	← (Reg$_1$,Reg$_2$)	1	store	r$_2$	⇒	r$_1$
3	Assign	→	← (+ (Reg$_1$,Reg$_2$),Reg$_3$)	1	storeAO	r$_3$	⇒	r$_1$,r$_2$
4	Assign	→	← (+ (Reg$_1$,Num$_2$),Reg$_3$)	1	storeAI	r$_3$	⇒	r$_1$,n$_2$
5	Assign	→	← (+ (Num$_1$,Reg$_2$),Reg$_3$)	1	storeAI	r$_3$	⇒	r$_2$,n$_1$
6	Reg	→	Lab$_1$	1	loadI	l$_1$	⇒	r$_{new}$
7	Reg	→	Val$_1$	0				
8	Reg	→	Num$_1$	1	loadI	n$_1$	⇒	r$_{new}$
9	Reg	→	◆ (Reg$_1$)	1	load	r$_1$	⇒	r$_{new}$
10	Reg	→	◆ (+ (Reg$_1$,Reg$_2$))	1	loadAO	r$_1$,r$_2$	⇒	r$_{new}$
11	Reg	→	◆ (+ (Reg$_1$,Num$_2$))	1	loadAI	r$_1$,n$_2$	⇒	r$_{new}$
12	Reg	→	◆ (+ (Num$_1$,Reg$_2$))	1	loadAI	r$_2$,n$_1$	⇒	r$_{new}$
13	Reg	→	◆ (+ (Reg$_1$,Lab$_2$))	1	loadAI	r$_1$,l$_2$	⇒	r$_{new}$
14	Reg	→	◆ (+ (Lab$_1$,Reg$_2$))	1	loadAI	r$_2$,l$_1$	⇒	r$_{new}$
15	Reg	→	+ (Reg$_1$,Reg$_2$)	1	add	r$_1$,r$_2$	⇒	r$_{new}$
16	Reg	→	+ (Reg$_1$,Num$_2$)	1	addI	r$_1$,n$_2$	⇒	r$_{new}$
17	Reg	→	+ (Num$_1$,Reg$_2$)	1	addI	r$_2$,n$_1$	⇒	r$_{new}$
18	Reg	→	+ (Reg$_1$,Lab$_2$)	1	addI	r$_1$,l$_2$	⇒	r$_{new}$
19	Reg	→	+ (Lab$_1$,Reg$_2$)	1	addI	r$_2$,l$_1$	⇒	r$_{new}$
20	Reg	→	- (Reg$_1$,Reg$_2$)	1	sub	r$_1$,r$_2$	⇒	r$_{new}$
21	Reg	→	- (Reg$_1$,Num$_2$)	1	subI	r$_1$,n$_2$	⇒	r$_{new}$
22	Reg	→	- (Num$_1$,Reg$_2$)	1	rsubI	r$_2$,n$_1$	⇒	r$_{new}$
23	Reg	→	× (Reg$_1$,Reg$_2$)	1	mult	r$_1$,r$_2$	⇒	r$_{new}$
24	Reg	→	× (Reg$_1$,Num$_2$)	1	multI	r$_1$,n$_2$	⇒	r$_{new}$
25	Reg	→	× (Num$_1$,Reg$_2$)	1	multI	r$_2$,n$_1$	⇒	r$_{new}$

Figure 11.4. Rewrite Rules for Tiling the Low-Level Tree with iloc.

Consider rule 16, which corresponds to the tree drawn in the margin. (Its result, at the + node, is implicitly a Reg.) The rule describes a tree that computes the sum of a value located in a Reg and an immediate value in a Num. The left side of the table gives the tree pattern for the rule, Reg → + (Reg$_1$,Num$_2$). The center column lists its cost, one. The right column shows an iloc operation that implements the rule, addI

r_1, n_2 🯄 rnew. The operands in the tree pattern, Reg_1 and Num_2, correspond to the operands r_1 and n_2 in the code template. The compiler must rewrite the field rnew in the code template with the name of a register allocated to hold the result of the addition. This register name will, in turn, become a leaf in the subtree that connects to this subtree. Notice that rule 16 has a commutative variant, rule 17. An explicit rule is needed to match subtrees such as the one drawn in the margin.

The rules in Figure 11.4 form a tree grammar similar to the grammars that we used to specify the syntax of programming languages. Each rewrite rule, or production, has a nonterminal symbol as its left-hand side. In rule 16, the nonterminal is Reg. Reg represents a collection of subtrees that the tree grammar can generate, in this case using rules 6 through 25. The right-hand side of a rule is a linearized tree pattern. In rule 16, that pattern is + (Reg_1,Num_2), representing the addition of two values, a Reg and a Num.

The rules in Figure 11.4 use Reg as both a terminal and a nonterminal symbol in the rules set. This fact reflects an abbreviation in the example. A complete set of rules would include a set of productions that rewrite Reg with a specific register name, such as Reg → r_0, Reg → r_1, …, and Reg → rk.

The nonterminals in the grammar allow for abstraction. They serve to connect the rules in the grammar. They also encode knowledge about where the corresponding value is stored at runtime and what form it takes. For example, Reg represents a value produced by a subtree and stored in a register, while Val represents a value already stored in register. A Val might be a global value, such as the arp. It might be the result of a computation performed in a disjoint subtree—a common subexpression.

The cost associated with a production should provide the code generator with a realistic estimate of the runtime cost of executing the code in the template. For rule 16, the cost is one to reflect the fact that the tree can be implemented with a single operation that requires just one cycle to execute. The code generator uses the costs to choose among the possible alternatives. Some matching techniques restrict the costs to numbers. Others allow costs that vary during matching to reflect the impact of previous choices on the cost of the current alternatives.

Tree patterns can capture context in a way that the simple treewalk code generator cannot. Rules 10 through 14 each match two operators (🯄 and +). These rules express the conditions in which the iloc operators loadAO and loadAI can be used. Any subtree that matches one of these five rules can be tiled with a combination of other rules. A subtree that matches rule 10 can also be tiled with the combination of rule 15 to produce an address and rule 9 to load the value. This flexibility makes the set of rewrite rules ambiguous. The ambiguity reflects the fact that the target machine has several ways to implement this particular subtree. Because the treewalk code generator matches one operator at a time, it cannot directly generate either of these iloc operations.

To apply these rules to a tree, we look for a sequence of rewriting steps that reduces the tree to a single symbol. For an ast that represents a complete program, that symbol should be the goal symbol. For an interior node, that symbol typically represents the value produced by evaluating the subtree rooted at the expression. The symbol also must specify where the value exists—typically in a register, in a memory location, or as a known constant value.

Figure 11.5 shows a rewrite sequence for the subtree that references the variable c in Figure 11.3. (Recall that c was at offset 12 from the label @G.) The leftmost panel shows the original subtree. The remaining panels show one reduction sequence for that subtree. The first match in the sequence recognizes that the left leaf (a Lab node) matches rule 6. This allows us to rewrite it as a Reg. The rewritten tree now matches the right-hand side of rule 11, □ (+ (Reg_1,Num_2)), so we can rewrite the entire subtree rooted at □ as a Reg. This sequence, denoted □6,11□, reduces the entire subtree to a Reg.

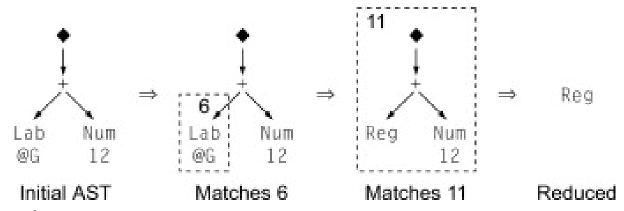


Figure 11.5. A Simple Tree Rewrite Sequence.

To summarize such a sequence, we will use a drawing like the one shown in the margin. The dashed boxes show the specific right-hand sides that matched the tree, with the rule number recorded in the upper left corner of each box. The list of rule numbers below the drawing indicates the sequence in which the rules were applied. The rewrite sequence replaces the boxed subtree with the final rule's left-hand side.

Notice how the nonterminals ensure that the operation trees connect appropriately at the points where they overlap. Rule 6 rewrites a Lab as a Reg. The left leaf in rule 11 is a Reg. Viewing the patterns as rules in a grammar folds all of the considerations that arise at the boundaries between operation trees into the labelling of nonterminals.

For this trivial subtree, the rules generate many rewrite sequences, reflecting the ambiguity of the grammar. Figure 11.6 shows eight of these sequences. All the rules in our scheme have a cost of one, except for rules 1 and 7. Since none of the rewrite sequences use these rules, their costs are equal to their sequence length. The sequences fall into three categories by cost. The first pair of sequences, □6,11□ and □8,14□, each have cost two. The next four sequences, □6,8,10□, □8,6,10□, □6,16,9□, and ⟨8,19,9⟩, each have cost three. The final sequences, ⟨6,8,15,9⟩ and ⟨8,6,15,9⟩, each have cost four.

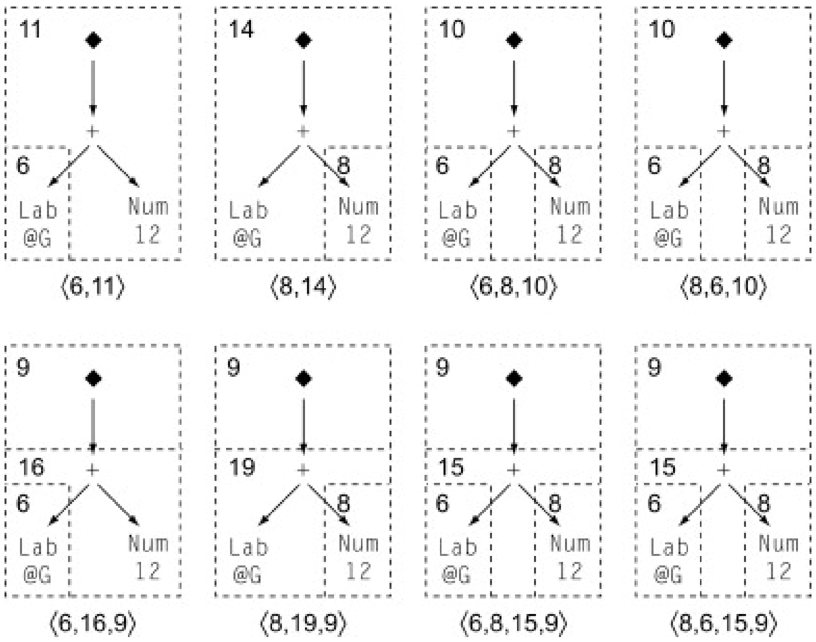


Figure 11.6. Potential Matches.

To produce assembly code, the selector uses the code templates associated with each rule. A rule's code template consists of a sequence of assembly-code operations that implements the subtree generated by the production. For example, rule 15 maps the tree pattern + (Reg_1,Reg_2) to the code template add r_1, r_2 → rnew. The selector replaces each of r_1 and r_2 with the register name holding the result of the corresponding subtree. It allocates a new virtual register name for rnew. A tiling for an ast specifies which rules the code generator should use. The code generator uses the associated templates to generate assembly code in a bottom-up walk. It supplies names, as needed, to tie the storage locations together and emits the instantiated operations corresponding to the walk.

The instruction selector should choose a tiling that produces the lowest-cost assembly-code sequence. Figure 11.7 shows the code that corresponds to each potential tiling. Arbitrary register names have been substituted where appropriate. Both ⟨6,11⟩ and ⟨8,14⟩ produce the lowest cost—two. They lead to different, but equivalent code sequences. Because they have identical costs, the selector is free to choose between them. The other sequences are, as expected, more costly.

```
loadI   @G      ⇒ ri
loadAI  ri,12   ⇒ rj
```
⟨6,11⟩

```
loadI   12      ⇒ ri
loadAI  ri,@G   ⇒ rj
```
⟨8,14⟩

```
loadI   @G      ⇒ ri
loadI   12      ⇒ rj
loadAO  ri,rj   ⇒ rk
```
⟨6,8,10⟩

```
loadI   12      ⇒ ri
loadI   @G      ⇒ rj
loadAO  ri,rj   ⇒ rk
```
⟨8,6,10⟩

```
loadI @G        ⇒ ri
addI  ri,12     ⇒ rj
load  rj        ⇒ rk
```
⟨6,16,9⟩

```
loadI 12        ⇒ ri
addI  ri,@G     ⇒ rj
load  rj        ⇒ rk
```
⟨8,19,9⟩

```
loadI @G        ⇒ ri
loadI 12        ⇒ rj
add   ri,rj     ⇒ rk
load  rk        ⇒ rl
```
⟨6,8,15,9⟩

```
loadI 12        ⇒ ri
loadI @G        ⇒ rj
add   ri,rj     ⇒ rk
load  rk        ⇒ rl
```
⟨8,6,15,9⟩

Figure 11.7. Code Sequences for the Matches.

If loadAI only accepts arguments in a limited range, the sequence ⟨8,14⟩ might not work, since the address that eventually replaces @G may be too large for the immediate field in the operation. To handle this kind of restriction, the compiler writer can introduce into the rewriting grammar the notion of a constant with an appropriately limited range of values. It might take the form of a new terminal symbol that can only represent integers in a given range, such as $0 \le i < 4096$ for a 12-bit field. With such a distinction, and code that checks each instance of an integer to classify it, the code generator could avoid the sequence ⟨8,14⟩, unless @G falls in the allowable range for an immediate operand of loadAI.

The cost model drives the code generator to select one of the better sequences. For example, notice that the sequence ⟨6,8,10⟩ uses two loadI operations, followed by a loadAO. The code generator prefers the lower-cost sequences, each of which avoids one of the loadI operations and issues fewer operations. Similarly, the cost model avoids the four sequences that use an explicit addition—preferring, instead, to perform the addition implicitly in the addressing hardware.

> Read full chapter