Definition Table

$Revision: 1.12 $

Compiler Tools Group
Department of Electrical and Computer Engineering
University of Colorado
Boulder, CO, USA
80309-0425

Table of Contents

The definition table is a data base in which the compiler stores information about defined entities like types, variables and procedures. Each entity is represented by a unique *definition table key*. Information about that entity is stored as an arbitrary number of *properties* associated with its definition table key. The value of a particular property can be set via an *update* operation and examined via a *query* operation. The definition table module exports an operation that yields a new definition table key and a distinguished key to represent an undefined entity that has no properties.

A user obtains a definition table module for a particular application by specifying the set of properties to be stored by that module. Different information may be associated with various kinds of entities, and specific items of information are determined at different times by various parts of the compiler. These characteristics determine the best ways of grouping individual items of information into properties, and it is possible to state general definition table design criteria based upon them.

The definition table module provides standard query and update operations. Additional operations are available from a library, and the user is allowed to define still others for particular applications. All of these operations use the same interface for accessing the definition table.

1 The Definition Table Module

The definition table module embodies the concept of a set of distinguishable entities, each having some set of properties. There is at least one entity, the invalid entity; it has an empty set of properties. No other entities exist unless they are explicitly created. Nothing is assumed about entities, other than the fact that they are distinguishable.

Each distinct entity is represented by a distinct key. The module exports a key representing the invalid entity, and an operation that creates a new key each time it is invoked.

Properties are declared by the user. Each property declaration specifies the type of data item that defines the property, but does not associate the property with any specific entity or entities.

Query and update operations are used to associate property values with entities. Each query or update operation is defined for a single property, and an invocation associates a single value of the declared type with the entity to which the operation is applied.

To make the definition table module accessible to a program, include the header file '`deftbl.h`' in that program.

1.1 How to create and use definition table keys

Definition table keys are objects of type `DefTableKey`. `DefTableKey` is a private type of the definition table module: Clients of the definition table module may declare variables and parameters of type `DefTableKey`, but they must make no assumptions about its representation.

The invalid key is `NoKey`, a legal value of type `DefTableKey`. No property values are ever associated with `NoKey`.

`NewKey` is a parameterless function that yields a value of type `DefTableKey` each time it is invoked. All of these values are distinct from each other and from `NoKey`. Any client of the definition table module may invoke `NewKey`.

Definition table keys are often bound to identifiers via the environment module operation `DefineIdn` (see Section 'Contour-Model Environment Module' in Library Reference Manual). In this case `DefineIdn` will invoke `NewKey` if necessary; `NewKey` should be invoked directly *only* for entities that are not under control of the environment module.

It is sometimes useful to be able to get a new key that is just like an existing one. The `CloneKey` operation takes a single key as argument, uses `NewKey` to get a new one, and initialises the properties of the new key to be the same as those of the key argument. The new key is returned.

Note that `CloneKey` implements shallow-copying of property values. For example, if a property value of the key argument is a pointer then after the `CloneKey` call two keys will have properties pointing to the same data.

1.2 How to declare properties

Each property has a *name* and a *type*.

A property type is denoted by a C identifier (a sequence of letters, digits and underscores that does not begin with a digit). Property types must either be built-in types of C (such as `int`), or they must be declared via `typedef` in some accessible module. The definition table module then becomes a client of the module defining the property type.

A property name is also denoted by a C identifier. Property names must be unique, and must be declared in a file of type `.pdl`. The simplest form of declaration is:

```
Name: Type;
```

Here `Name` is the property name being declared and `Type` is the data type of the possible values for that property.

Any type that can be returned by a function can be used as the type of a property. If the type is declared via `typedef`, some `.pdl` file must contain a C string (sequence of characters delimited by `"` characters) that specifies the name of the file containing that declaration.

1.3 Behavior of the basic query operations

Each declared property has a basic query operation. If the name declared for the property is `Name`, then the basic query operation is a function named `GetName`. If property `Name` has been declared to have values of type `Type`, then the function implementing the basic query operation has the following prototype:

```
Type GetName(DefTableKey key, Type deflt)
```

If `GetName` is applied to a definition table key with which a value of the `Name` property has been associated, then `GetName` returns the associated value. Otherwise it returns the value of parameter `deflt`.

Since `NoKey` represents an invalid entity that never has associated property values, applying any basic query operation to `NoKey` will yield the value of parameter `deflt`.

1.4 Behavior of the basic update operations

Each declared property is has two basic update operations. If the name declared for the property is `Name`, then the basic update operations are functions named `SetName` and `ResetName`. If property `Name` has been declared to have values of type `Type`, then the functions implementing the basic update operations have the following prototypes:

```
void SetName(DefTableKey key, Type add, Type replace)
void ResetName(DefTableKey key, Type val)
```

If `SetName` is applied to a definition table key with which a value of the `Name` property has been associated, then that value is replaced by the value of parameter `replace`. Otherwise the value of parameter `add` becomes the value of the `Name` property associated with that definition table key. Application of `ResetName` to a definition table key always results in the value of the `Name` property being set to `val`.

Since `NoKey` represents an invalid entity that has no associated property values, applying any basic update operation to `NoKey` has no effect.

1.5 A simple definition table application

Defining occurrences of identifiers are the points at which those identifiers are declared, while *applied occurrences* are points at which they are used. In many programming languages, it is possible to distinguish defining occurrences from applied occurrences on the basis of context. Let us assume that this is the case, and use the definition table to verify that each identifier has exactly one defining occurrence.

The environment module is used to implement the scope rules of the language, binding a definition table key to each occurrence of an identifier. Within each individual scope, the same key will be bound to all occurrences of a particular identifier. To verify that there is a single defining occurrence, associate a property `Def` with the definition table key.

`Def` is of type integer, and three values are significant:

`0` There is no defining occurrence

`1` There is exactly one defining occurrence

`2` There is more than one defining occurrence

At each defining occurrence, the update operation `SetDef` is invoked with the `add` parameter `1` and the `replace` parameter `2`. After all defining occurrences are processed, the `Def` property value `1` will be associated with the definition table key for each identifier having exactly one defining occurrence. The `Def` property value `2` will be associated with the definition table key for each identifier having more than one defining occurrence, and there won't be any `Def` property value associated with the definition table key for each identifier without defining occurrences.

At each applied occurrence, the query operation `GetDef` is invoked with the `deflt` parameter `0`. If the identifier has one or more defining occurrences, `GetDef` will yield the `Def` property value (either `1` or `2`) associated with the definition table key for the applied occurrence. Otherwise there will be no `Def` property value associated with the definition table key for the applied occurrence, and `GetDef` will yield the value of the `deflt` parameter: `0`.

2 Definition Table Design Criteria

There are many properties that might be of interest to a compiler. A Pascal compiler needs to know the type and value of a constant. More information is needed for a variable: its type, the static nesting level of the procedure containing its declaration, and where it is located in the target machine's memory. The compiler designer must decide how to represent this information.

The first task in definition table design is to select the set of entities to be represented by definition table keys. Then a set of properties must be defined to carry the information associated with the entities. There is no need to specify relationships between properties and entities: a value of any property may be associated with any entity.

2.1 Criteria for selecting entities

Identifier definitions must be represented by definition table keys if the normal environment module is used. Whether there are other entities that should be represented by definition table keys depends on the particular translation problem.

Entities that are invariably created by an identifier declaration can be represented by the definition table key bound to that declaration. If an entity may be created without being bound to an identifier, however, then it must be represented by a distinct definition table key. For example, the following Pascal variable declaration defines an identifier and creates a variable bound to that identifier, but also creates a type that is not bound to any identifier:

```
var I: 1..10;
```

A Pascal compiler could use one key for the definition of `I` and the variable, since Pascal variables are created only in conjunction with identifier declarations. It must use a separate key for the subrange type, however, since types can be created without declaring any identifier.

Distinct definition table keys should be used for generated entities that are "similar" to user-defined entities. For example, Pascal labels are created only in conjunction with label definitions; a user-defined label entity can therefore be represented by the definition table key requested by the environment module to represent the label definition. A Pascal compiler will probably create labels in the course of translating statements like `if`, `case` and `while`. These labels should also be represented by definition table keys (obtained from `NewKey`) to maintain compatibility with user-defined labels. This compatibility is important because of the semantics common to generated and user-defined labels. If it is necessary to distinguish the two, that is easily done via a Boolean property.

2.2 Criteria for grouping data values

Each definition table key provides access to information about an entity. The information is embodied in a set of properties. How should those properties be defined?

One possibility is to define a single property to carry all of the information associated with a particular definition table entry. That property would be a structure, with distinct fields to hold the distinct items of information. There is a subtle problem with this approach: Because the items of information associated with an entity are determined at different times by different modules, fields of the structure will be undefined for various periods during its lifetime. If, through an oversight in the design, the compiler accesses one of these undefined fields, then it may very well crash. Such errors are often extremely difficult to diagnose, and the compiler development time is thus increased unnecessarily.

A better approach is to group related information that is obtained at a particular point in the compilation as a single property, and to leave unrelated information or information that is obtained at several different points as separate properties. When a property value is set, that value is complete (if any information was not available at the time, it would have been treated as a separate property), and any query must supply a consistent default value to be returned in case the desired property is not available.

In general, it is better to err on the side of too many properties than too few. Each definition table key is actually implemented as a pointer to a list, with the properties being elements of that list. The list element consists of a code identifying the property and a block of storage large enough to hold a value of the property's type. The length of the list for a particular definition table key is the number of values that have actually been associated with that definition table key. If no update operation is performed for a particular property on an entity, nothing is stored for that property. A valid value is guaranteed to be returned from a query operation because of the default argument supplied to the query call. A default value for a property can be simulated by always using the same value every time the query operation is used for that property.

When properties are combined, the number of list entries may be reduced. (This is not always the case, because two distinct properties only require one list element if one of these properties has its default value.) If the number of list entries is reduced, the time to access properties is reduced. Normally, however, property lists are short and the time to access properties is an insignificant fraction of the total processing time. Thus there is usually little payoff in access time from combining properties.

3 The Definition Table Interface

The interface to the definition table module has two parts, one fixed and the other dependent on a specification supplied by the user. The fixed part of the interface exports the value `NoKey` and the operation `NewKey` (see Section 1.1 [How to create and use definition table keys], page 2). The variable part of the interface exports the query and update operations for the properties specified by the user.

A library of predefined query and update operations is provided to implement common tasks; users can also provide their own operations. The set of operations and properties for a specific processor is defined by a specification written in a special-purpose language.

PDL generates definitions for each of the operations specified in files of type '`.pdl`'. These definitions are made available in the generated file '`pdl_gen.h`'. Although this file is automatically included for use in your attribute grammar specifications, any C files which use definition table operations must include this file.

3.1 Predefined query and update operations

The basic query and update operations for the `Name` property are `GetName` (see Section 1.3 [Behavior of the basic query operations], page 3), `SetName`, and `ResetName` (see Section 1.4 [Behavior of the basic update operations], page 4). These operations are sufficient in most cases, and are provided automatically for every property. Other operations, such as `IsName` and `UniqueName` are available, but must be explicitly requested as discussed in the next section.

```
void IsName(DefTableKey key, Type which, Type error)
```

If `IsName` is applied to a definition table key that has no associated `Name` property, then a `Name` property with the value of parameter `which` becomes associated with that definition table key as a result of the operation. If it is applied to a definition table key that does have an associated `Name` property, and the current value of that property is not equal to the value of the parameter `which`, then the value of that property is changed to the value of the parameter `error`. Otherwise the operation has no effect.

Since `NoKey` represents an invalid entity that has no properties, applying `IsName` to `NoKey` has no effect.

```
void UniqueName(DefTableKey key, Type next())
```

If `UniqueName` is applied to a definition table key that has no associated `Name` property, then a `Name` property with the value returned by the invocation of parameter `next` becomes associated with that definition table key as a result of the operation. Otherwise the operation has no effect.

Since `NoKey` represents an invalid entity that has no properties, applying `UniqueName` to `NoKey` has no effect.

Parameter `next` is invoked if and only if `UniqueName` is applied to a definition table key that differs from `NoKey` and has no associated `Name` property.

3.2 The property definition language

The property definition language allows a user to specify an arbitrary set of properties of arbitrary types, to assert that certain operations from the library should be available to query or update these properties, and to define new operations. It also allows a user to establish the initial state of the definition table. Specifications in the property definition language are distinguished by being provided in files of type `pdl`.

An arbitrary number of such files may be provided; they will be concatenated to form the complete specification of the variable part of the definition table module's interface. Each file consists of a set of property and operation declarations as are described in the following sections. Because the files are concatenated, specifications in one need not be repeated in another. Nevertheless, we strongly suggest that each file contain a complete specification for one or more tasks. This allows maximum reuse of existing text.

C pre-processor directives and C comments can be used in type-`pdl` files.

3.2.1 How to declare properties

Properties are declared by specifying the property name and type, optionally with a set of operations that should apply to properties of the type specified. If the type is defined by a typedef, and is not equal to `DefTableKey`, the file containing the typedef must be specified.

The general form of a property declaration is given by:

```
PropertySpec: FileName / PropertyDecl .
FileName: String .
PropertyDecl: PropertyNameList ':' Type ';' .
PropertyNameList: Identifier / PropertyNameList ',' Identifier .
Type: Identifier / Identifier '[' OperationList ']' .
OperationList: Identifier / OperationList ',' Identifier .
```

Both `String` and `Identifier` are constructed according to the rules of C. The `FileName` string must be a valid file name. Each `Identifier` appearing in a `PropertyNameList` is a defining occurrence; all other occurrences of `Identifier` are applied. Multiple defining occurrences for property names are allowed, provided that they all define the property to hold values of the same type.

Operation names are formed by concatenating an `Identifier` appearing in an `OperationList` with an `Identifier` appearing in a `PropertyNameList`. `Reset`, `Get` and `Set` are automatically defined for every property, and need not appear in any `OperationList`.

Here are some valid property definitions:

```
Def, Kind: int;
Type: DefTableKey [Is];
Storage: StorageRequired; "Storage.h"
```

`Def` and `Kind` are integer-valued properties. The variable part of the definition table interface will export operations `GetDef`, `SetDef`, `ResetDef`, `GetKind`, `SetKind`, and `ResetKind`. It will also export `GetType`, `SetType`, `ResetType`, and `IsType` because the library operation `Is` appears in the `OperationList` on the second line. Note that this specification will produce `GetStorage`, `SetStorage`, and `ResetStorage`, but will not produce `IsStorage`, because `Is` does not appear in any `OperationList` for the property `Storage`. If `Is` did appear in an `OperationList` for the property `Storage` anywhere in the specification, even in another type-`pdl` file, then the generated module would export `IsStorage`.

Type `int` is a primitive type of C, and `DefTableKey` is defined by the definition table module itself. Thus neither of these types needs to be defined specially. `StorageRequired` must be defined, however, and therefore file '`storage.h`' is named explicitly (see Section 'Storage Allocation Module' in Library Reference Manual). This file name could be placed anywhere in the specification, even in another type-`pdl` file. The order in which such header files are named within any given `pdl` file is maintained in the generated modules.

3.2.2 How to declare operations

Operations can be declared by specifying a name, a prototype and a body. The operations are generic, with the operand and result types depending on the type of the property for which the generic operation is associated (see Section 3.2.1 [How to declare properties], page 10). If an operation is declared to have the same name as one present in the library, the user-defined operation will take precedence.

The general form of an operation declaration is given by:

```
OperationDecl: Gtype Identifier '(' Parameters ')' Text .
Gtype: 'TYPE' / Ctype .
Parameters: Parameter / Parameters ',' Parameter .
Parameter: Gtype Identifier .
```

`TYPE` is used to represent the type of the property with which the operation is associated, while `Ctype` stands for any valid C type declarator. One of the parameters (by convention the first) must be of type `DefTableKey`, and must have the name `key`; the other parameters are arbitrary.

`Text` is any C compound statement, enclosed in `{}`. Within this compound statement, certain macros may be used:

`PRESENT` Returns true is the property has an associated value and false if it does not.

`ACCESS` The return value is the same as that of PRESENT, but ACCESS guarantees that space has been allocated for the property after invocation.

`VALUE` Current value of the property.

The `VALUE` macro can be used either as the source of an existing value or the destination for a new value. It is defined after an invocation of the `ACCESS` macro, or whenever the `PRESENT` macro returns true.

Here is the declaration of the basic query operation from the library:

```
TYPE Get(DefTableKey key, TYPE deflt)
{ if (key == NoKey) return deflt;
  if (PRESENT) return VALUE;
  else return deflt;
}
```

The type of the value returned by a `Get` operation is the type of the associated property (`TYPE`), which is also the type of the deflt parameter. PRESENT is used to check whether a value is associated with the property, and if so that value (VALUE) is returned.

Here is the declaration of the Set operation from the library:

```
void Set(DefTableKey key, TYPE add, TYPE replace)
{ if (key == NoKey) return;
  if (ACCESS) VALUE = replace;
  else VALUE = add;
}
```

No value is returned by a Set operation. ACCESS is used to check whether a value is associated with the property, and also to guarantee that space for a value is available. The available space is then filled appropriately.

Here is the declaration of the Reset operation from the library:

```
void Reset(DefTableKey key, TYPE val)
{ if (key == NoKey) return;
  ACCESS; VALUE = val;
}
```

No value is returned by a Reset operation. ACCESS is used to ensure that space is made available to hold the value of the property. The value is then set to val.

Here is the declaration of the conditional update operation from the library:

```
void Is(DefTableKey key, TYPE which, TYPE error)
{ if (key == NoKey) return;
  if (!ACCESS) VALUE = which;
  else if (VALUE != which) VALUE = error;
}
```

Here is the library operation that guarantees a unique value for a property:

```
void Unique(DefTableKey key, TYPE next())
{ if (key == NoKey) return;
  if (!ACCESS) VALUE = next();
}
```

The next parameter is a function that delivers a new value of the type of the associated property each time it is called. It will be invoked only when there is currently no value associated with the property.

3.2.3 How to specify the initial state

The initial state of the definition table consists of a set of *known keys*, some of which may have associated property values. Each known key is represented by an identifier, which can be used anywhere that a value of type `DefTableKey` is required.

The general form of a known key specification is given by:

```
KnownKey: Identifier PropertyValueList ';' .
PropertyValueList: / '->' PropertyValues .
PropertyValues: PropertyValue // ',' .
PropertyValue: Identifier '=' Text .
```

`Text` is any C initializer valid for the type of the property, enclosed in `{}`. It may contain constant identifiers, including identifiers that represent known keys, regardless of where they are declared. Each `Identifier` appearing in a `PropertyValue` must be declared elsewhere in the PDL specification (i.e. in some type-`pdl` file, see Section 3.2 [How to declare properties], page 10).

Here are some valid specifications of known keys:

```
ErrorType;
IntegerKey -> Def={1}, Type={IntegerType};
IntegerType -> Storage = {4,4,0};
```

The known key `ErrorType` has no properties initially, while the known key `IntegerKey` has two and the known key `IntegerType` has one. All of these properties were declared above (see Section 3.2 [How to declare properties], page 10). `IntegerType` is a value of type `DefTableKey`, and is therefore a valid initializer for the `Type` property of `IntegerKey`. To see that the initializer given for the `Storage` property of `IntegerType` is valid, one would need to consult the file '`storage.h`'. That file is the interface specification for the data mapping module, and its name appeared in the declaration of the `Storage` property.

4 PDL Input Grammar

```
Source: Spec .
Spec: ( PropertyDecl / FileDecl / OperationDecl / KnownKeyDecl )* .

PropertyDecl: PropertyNameList ':' Type ';' .
PropertyNameList: Identifier / PropertyNameList ',' Identifier .
Type: Identifier / Identifier '[' OperationList ']' .
OperationList: Identifier / OperationList ',' Identifier .

FileDecl: String .

OperationDecl: Gtype Identifier '(' Parameters ')' Text .
Gtype: 'TYPE' / Ctype .
Parameters: Parameter / Parameters ',' Parameter .
Parameter: Gtype Identifier .

KnownKeyDecl: Identifier PropertyValueList ';' .
PropertyValueList: / '->' PropertyValues .
PropertyValues: PropertyValue // ',' .
PropertyValue: Identifier '=' Text .
```

In the above grammar, Text refers to a C block enclosed in braces.

Index