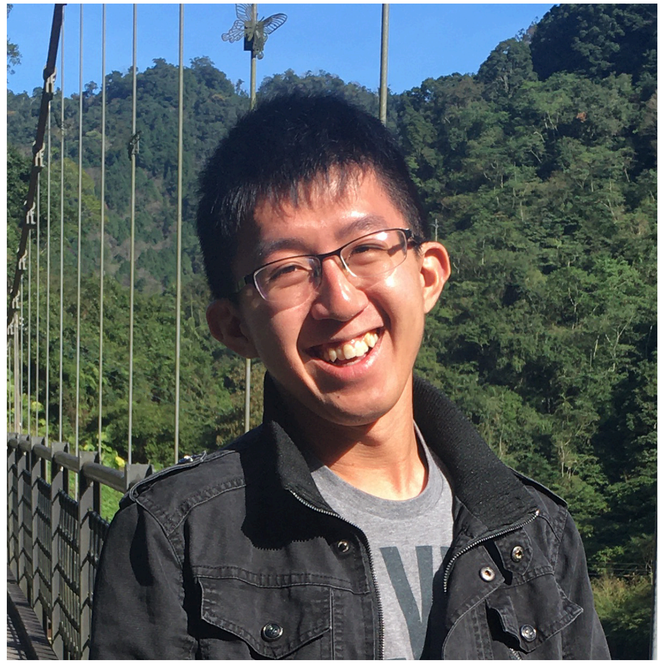
How to write a TableGen backend

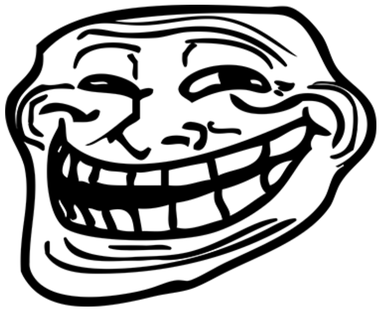
Min-Yih “Min” Hsu @ LLVM Dev Meeting 2021

$ whoami

“Min” Hsu

- Computer Science PhD Candidate in University of California, Irvine
- Code owner of M68k LLVM backend
- Author of book “*LLVM Techniques, Tips and Best Practices*” (2021)





TableGen Backend

TableGen in a nutshell

TableGen in a nutshell

- A **Domain-Specific Language (DSL)** originated from the LLVM project.
  - It’s Turing complete!

TableGen in a nutshell

- A **Domain-Specific Language (DSL)** originated from the LLVM project.
  - It’s Turing complete!
- Originally invented to describe the **instruction table** of a LLVM target.

TableGen in a nutshell

- A **Domain-Specific Language (DSL)** originated from the LLVM project.
  - It's Turing complete!
- Originally invented to describe the **instruction table** of a LLVM target.
  - Ex. The operands, assembly syntax, and ISel rules for each instruction.

TableGen in a nutshell

- A **Domain-Specific Language (DSL)** originated from the LLVM project.
  - It's Turing complete!
- Originally invented to describe the **instruction table** of a LLVM target.
  - Ex. The operands, assembly syntax, and ISel rules for each instruction.
- Now: Used in a wide variety of (completely) different areas inside LLVM.

TableGen in a nutshell

- A **Domain-Specific Language (DSL)** originated from the LLVM project.
  - It's Turing complete!
- Originally invented to describe the **instruction table** of a LLVM target.
  - Ex. The operands, assembly syntax, and ISel rules for each instruction.
- Now: Used in a wide variety of (completely) different areas inside LLVM.
  - Instruction scheduling info.

TableGen in a nutshell

- A **Domain-Specific Language (DSL)** originated from the LLVM project.
  - It's Turing complete!
- Originally invented to describe the **instruction table** of a LLVM target.
  - Ex. The operands, assembly syntax, and ISel rules for each instruction.
- Now: Used in a wide variety of (completely) different areas inside LLVM.
  - Instruction scheduling info.
  - Declaring IR attributes.

TableGen in a nutshell

- A **Domain-Specific Language (DSL)** originated from the LLVM project.
  - It's Turing complete!
- Originally invented to describe the **instruction table** of a LLVM target.
  - Ex. The operands, assembly syntax, and ISel rules for each instruction.
- Now: Used in a wide variety of (completely) different areas inside LLVM.
  - Instruction scheduling info.
  - Declaring IR attributes.
  - LLVM `Option` subsystem (e.g. Clang's compiler flags).

```
class Stuff {


}
```

```
class Stuff {

}
```

Layout

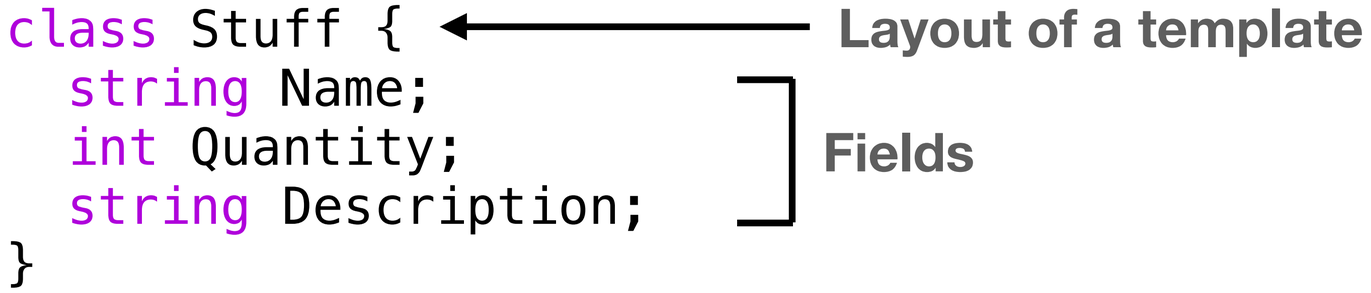
```
class Stuff {

}
```

Layout of a template

```
class Stuff {
  string Name;
  int Quantity;
  string Description;
}
```

Layout of a template

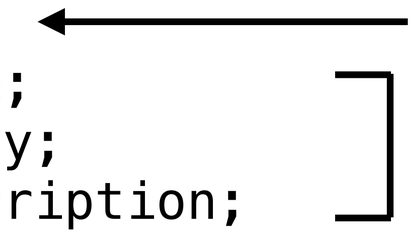

class Stuff {
string Name;
int Quantity;
string Description;
}
Layout of a template
Fields

```
class Stuff {
  string Name;
  int Quantity;
  string Description;
}

def water_bottle : Stuff {



}
```



Layout of a template

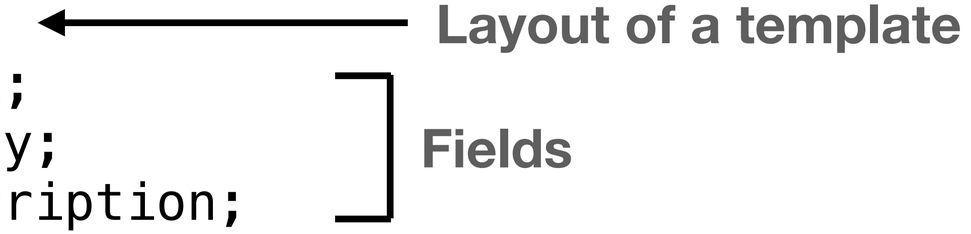
Fields

```
class Stuff {
  string Name;
  int Quantity;
  string Description;
}

def water_bottle : Stuff {



}
```



```
class Stuff {
  string Name;
  int Quantity;
  string Description;
}

def water_bottle : Stuff {


}
```

Layout of a template

Fields

A *record* created with template `Stuff`

```
class Stuff {
  string Name;
  int Quantity;
  string Description;
}

def water_bottle : Stuff {
  let Name = "Water bottle";
  let Quantity = 1;
  let Description = "A stuff that helps you hydrate.";
}
```

Layout of a template

Fields

A *record* created with template `Stuff`

```
class Stuff {
  string Name;
  int Quantity;
  string Description;
}

def water_bottle : Stuff {
  let Name = "Water bottle";
  let Quantity = 1;
  let Description = "A stuff that helps you hydrate.";
}

def smart_phone : Stuff {
  let Name "Smart phone";
  let Quantity = 2;
  let Description = "A stuff that keeps you from hydrating.";
}
```

Layout of a template

Fields

A *record* created with template `Stuff`

Another record created with template `Stuff`

```
class Stuff <string name, int quantity, string description> {
  string Name = name;
  int Quantity = quantity;
  string Description = description;
}
```

```
class Stuff <string name, int quantity, string description> {
  string Name = name;
  int Quantity = quantity;
  string Description = description;
}

def water_bottle : Stuff<"Water bottle", 1,
                         "A stuff that helps you hydrate.">;

def smart_phone : Stuff<"Smart phone", 2,
                        "A stuff that prevents you from hydrating.">;
```

OOP v.s. TableGen

```
class Person {
    std::string Name;
    int Age;
    JobKind Job;
};
Person Me{"Min", 12, WEEBUS};
```

OOP v.s. TableGen

```
class Person {
    std::string Name;
    int Age;
    JobKind Job;
};
Person Me{"Min", 12, WEEBUS};
```

✔ Encapsulating data

OOP v.s. TableGen

```
class Person {
    std::string Name;
    int Age;
    JobKind Job;
};
Person Me{"Min", 12, WEEBUS};

void foo(int N) {

  Me.Name = "Max";




}
```

✔ Encapsulating data

✘ Records are immutable

OOP v.s. TableGen

```
class Person {
    std::string Name;
    int Age;
    JobKind Job;
};
Person Me{"Min", 12, WEEBUS};

void foo(int N) {

  Me.Name = "Max";

  Person Rick{“Rick", N, SINGER};



}
```

- ✓ Encapsulating data
- ✗ Records are immutable
- ✗ Constant values in fields

OOP v.s. TableGen

```
class Person {
    std::string Name;
    int Age;
    JobKind Job;
};
Person Me{"Min", 12, WEEBUS};

void foo(int N) {

  Me.Name = "Max";

  Person Rick{“Rick", N, SINGER};

  for (i = 0; i < N; ++i)
    SomeList.emplace_back(Person{...});
}
```

- ✓ Encapsulating data
- ✗ Records are immutable
- ✗ Constant values in fields
- ✗ Constant numbers of records

Relational DB v.s. TableGen

Table “*Stuff*”

Name	Quantity	Description
Water Bottle	1	…
Smart Phone	2	…

Relational DB v.s. TableGen

Table “*Stuff*”

Name	Quantity	Description
Water Bottle	1	…
Smart Phone	2	…

✔ Fields are similar to columns

Relational DB v.s. TableGen

Table “*Stuff*”

Name	Quantity	Description
Water Bottle	1	…
Smart Phone	2	…

✓ Fields are similar to columns

✗ Records do not *belong* to any table

Relational DB v.s. TableGen

Table “*Stuff*”

Name	Quantity	Description
Water Bottle	1	…
Smart Phone	2	…

Fields are similar to columns

Records do not *belong* to any table

Strong structure is *not* required

Relational DB v.s. TableGen

Table “*Stuff*”

Name	Quantity	Description
Water Bottle	1	…
Smart Phone	2	…

Fields are similar to columns

Records do not *belong* to any table

Strong structure is *not* required

```
def foo {
  string A = “foo”;
  int B = 0;
}

def bar {
  int Z = 1;
}

def zoo;
```

TableGen data types

TableGen data types

- Primitive types (common): `int`, `string`, `bool`, `bit`

TableGen data types

- Primitive types (common): `int`, `string`, `bool`, `bit`
- Bit vector: `bits<`*N*`>`

TableGen data types

- Primitive types (common): `int`, `string`, `bool`, `bit`
- Bit vector: `bits<`*N*`>`
- List: `list<`*T*`>`

TableGen data types

- Primitive types (common): `int`, `string`, `bool`, `bit`
- Bit vector: `bits<`*N*`>`
- List: `list<`*T*`>`
- Direct Acyclic Graph (DAG): `dag`

TableGen data types

- Primitive types (common): `int`, `string`, `bool`, `bit`
- Bit vector: `bits<`*N*`>`
- List: `list<`*T*`>`
- Direct Acyclic Graph (DAG): `dag`
  - Represent DAG data *symbolically*

TableGen data types

- Primitive types (common): `int`, `string`, `bool`, `bit`
- Bit vector: `bits<`*`N`*`>`
- List: `list<`*`T`*`>`
- Direct Acyclic Graph (DAG): `dag`
  - Represent DAG data *symbolically*
  - `dag foo = (`*`operator arg0, arg1, …`*`)`

How to use TableGen records?

How to use TableGen records?

```
def water_bottle : Stuff<"Water bottle", 1,
                         "A stuff that…”>;

def smart_phone : Stuff<"Smart phone", 2,
                        "A stuff that…”>;
```

How to use TableGen records?

```
def water_bottle : Stuff<"Water bottle", 1,
                         "A stuff that…”>;

def smart_phone : Stuff<"Smart phone", 2,
                        "A stuff that…”>;
```

How to use TableGen records?

```
def water_bottle : Stuff<"Water bottle", 1,
                         "A stuff that…”>;

def smart_phone : Stuff<"Smart phone", 2,
                        "A stuff that…”>;
```

How to use TableGen records?

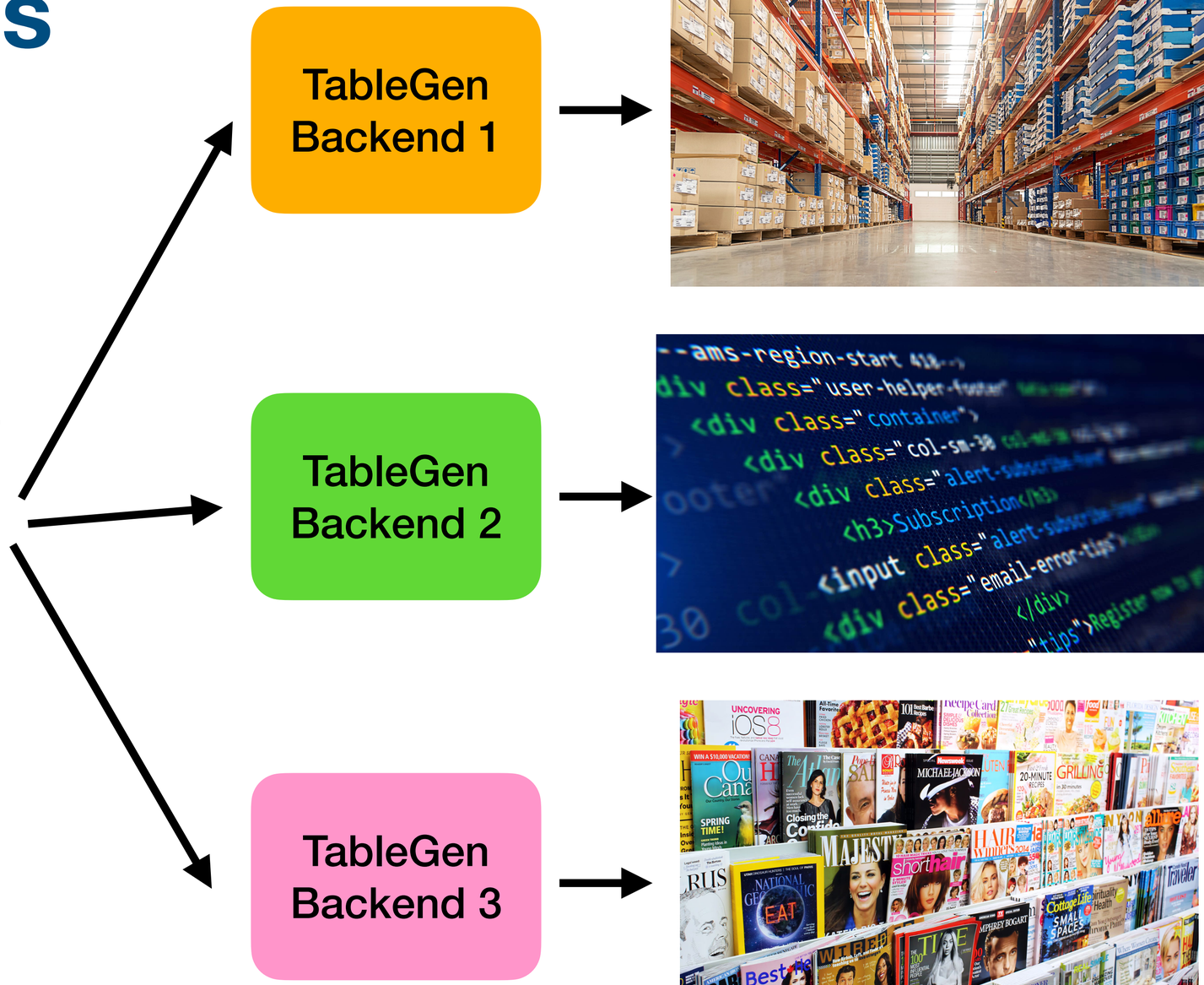
```
def water_bottle : Stuff<"Water bottle", 1,
                         "A stuff that…”>;

def smart_phone : Stuff<"Smart phone", 2,
                        "A stuff that…”>;
```

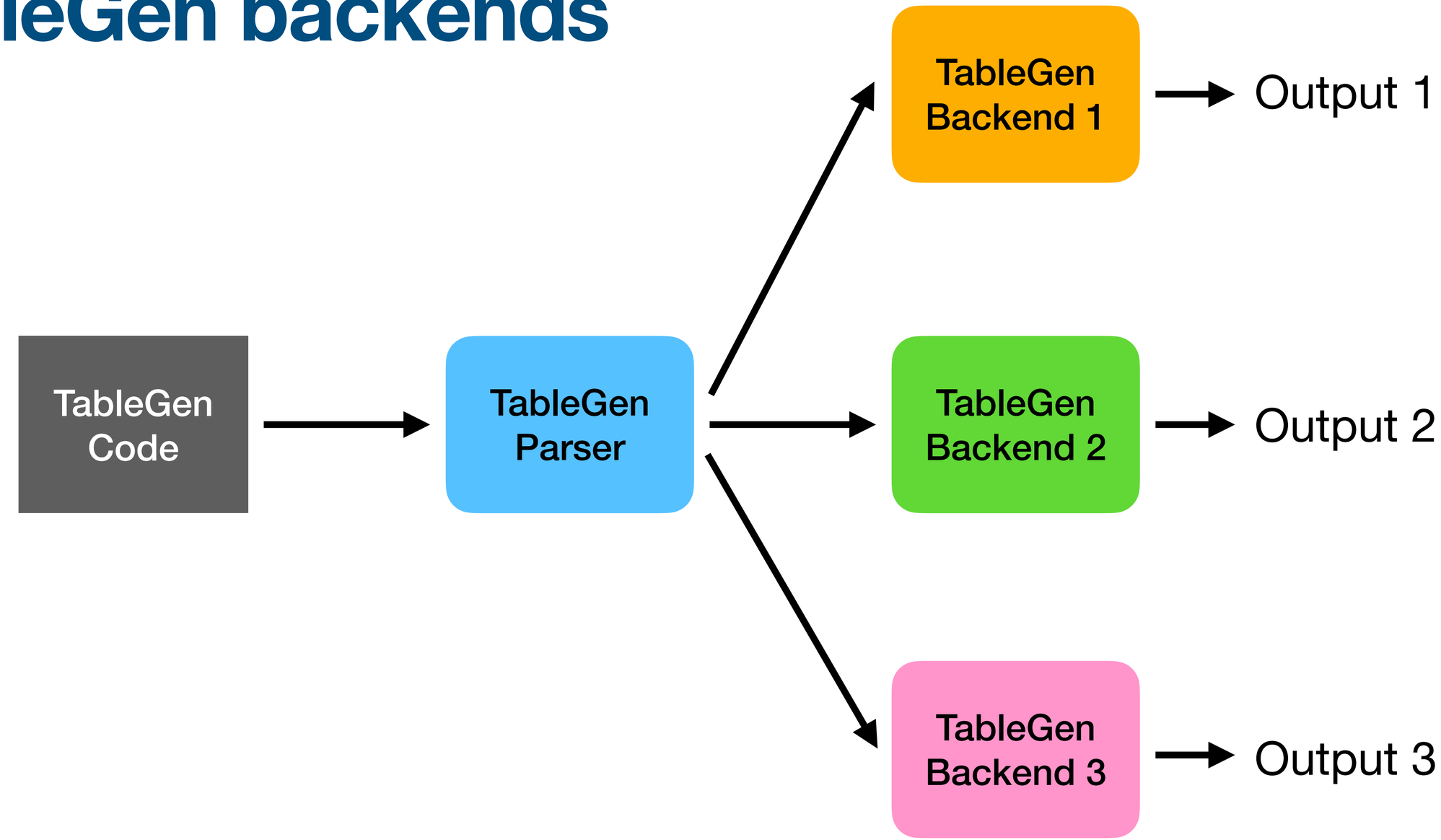
TableGen backends

```
def water_bottle : Stuff<"Water bottle", 1,
                         "A stuff that…”>;

def smart_phone : Stuff<"Smart phone", 2,
                        "A stuff that…”>;
```



TableGen backends



TableGen usage in LLVM

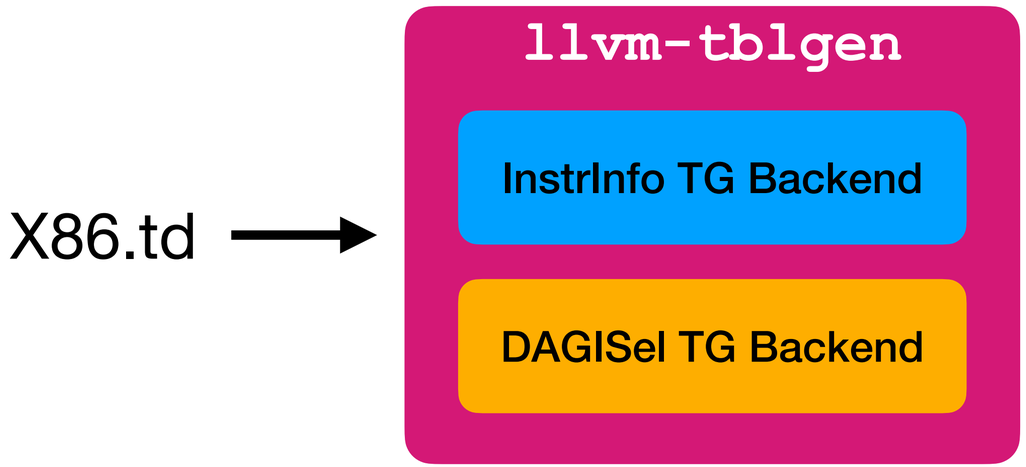
An example in LLVM backend

TableGen usage in LLVM

An example in LLVM backend

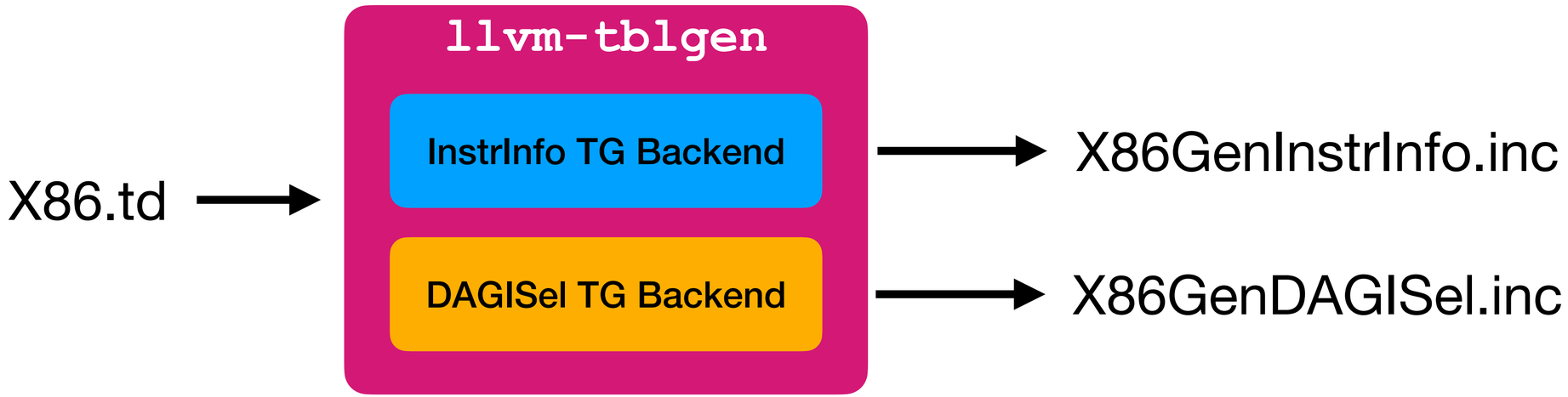
TableGen usage in LLVM

An example in LLVM backend



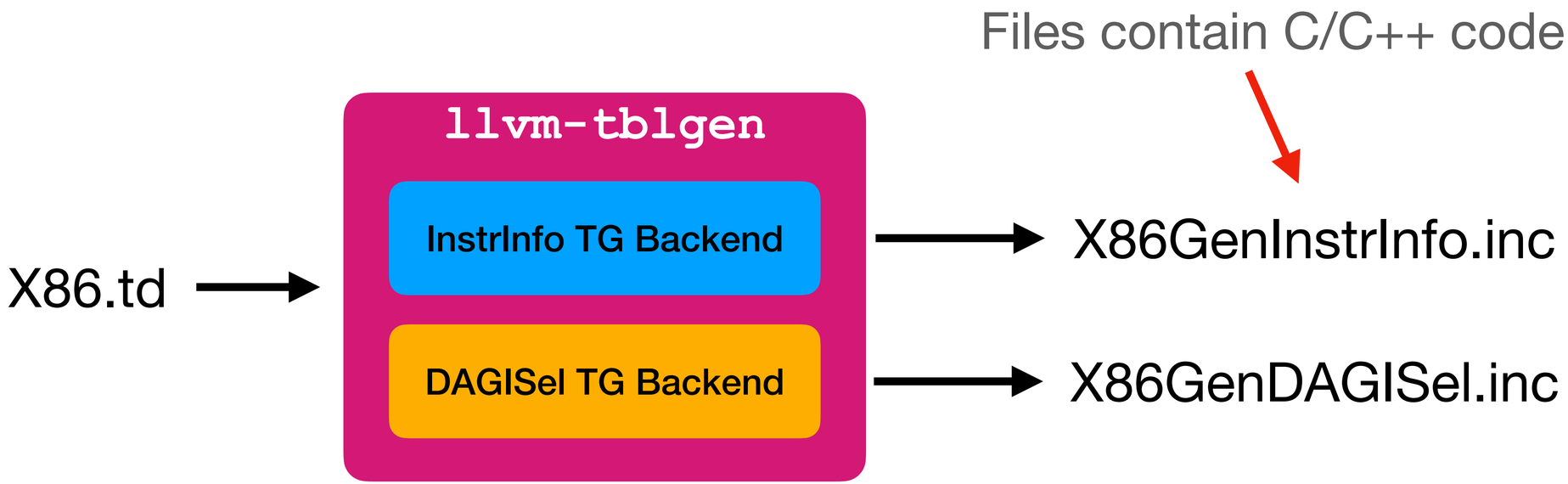
TableGen usage in LLVM

An example in LLVM backend



TableGen usage in LLVM

An example in LLVM backend



TableGen Backend Development

Why should I learn to write a TG backend?

Why should I learn to write a TG backend?

Despite being a DSL, TableGen is actually pretty **versatile**

Why should I learn to write a TG backend?

Despite being a DSL, TableGen is actually pretty **versatile**

Always require a specific TableGen backend

Why should I learn to write a TG backend?

Despite being a DSL, TableGen is actually pretty **versatile**

Always require a specific TableGen backend

LLVM has provided nice infrastructures to work with TableGen code

Project overview

Project overview

Recap: Comparison with Relational DB

Relational DB

Table "*Stuff*"

Name	Quantity	Description
Water Bottle	1	…
Smart Phone	2	…

v.s.

TableGen

- ✓ Fields are similar to columns
- ✗ Records do not *belong* to any table
- ▲ Strong structure is *not* required

Project overview

Recap: Comparison with Relational DB

A more *flexible* way to represent **static** structural data

Relational DB **v.s.** **TableGen**

Table "*Stuff*"

Name	Quantity	Description
Water Bottle	1	…
Smart Phone	2	…

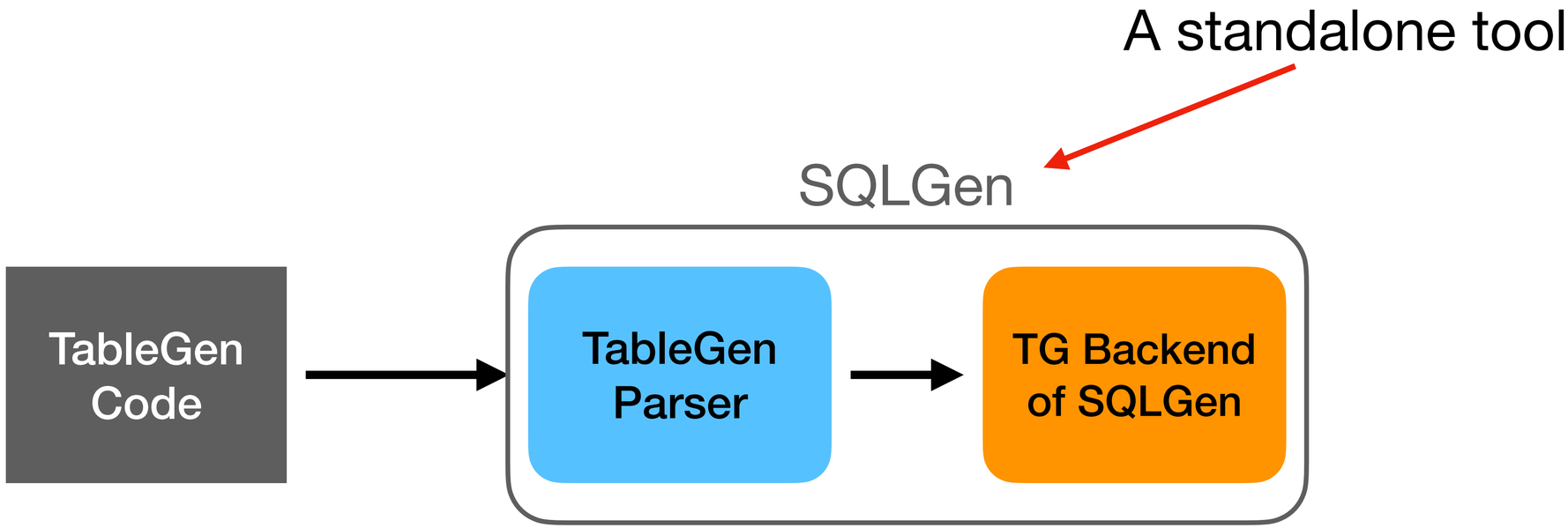
- ✓ Fields are similar to columns
- ✗ Records do not *belong* to any table
- ▲ Strong structure is *not* required

Project overview

SQLGen — Generate SQL from TableGen code

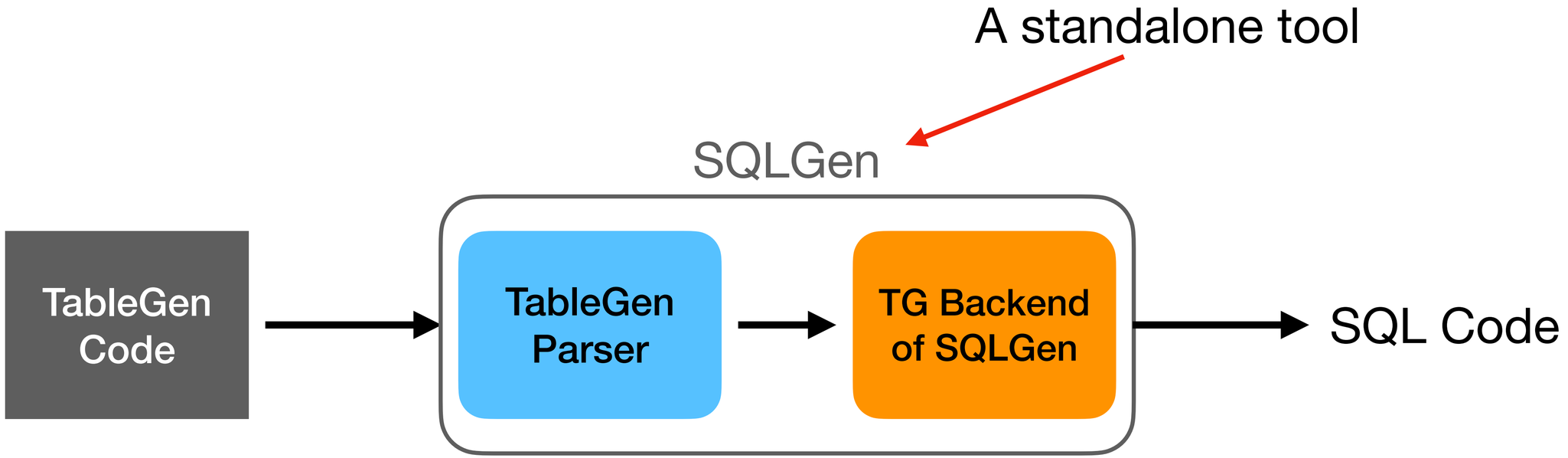
Project overview

SQLGen — Generate SQL from TableGen code



Project overview

SQLGen — Generate SQL from TableGen code



TableGen syntax in SQLGen

SQL table creation

Input TableGen Code

Generated SQL Code

TableGen syntax in SQLGen

SQL table creation

```
CREATE TABLE Customer (
    ID          int,
    Name        varchar(255),
    Affiliation varchar(255),
    PRIMARY KEY (ID)
);
```

Input TableGen Code

Generated SQL Code

TableGen syntax in SQLGen

SQL table creation

```
class Table {
  int PrimaryKey = 0;
}
```

Input TableGen Code

```
CREATE TABLE Customer (
    ID          int,
    Name        varchar(255),
    Affiliation varchar(255),
    PRIMARY KEY (ID)
);
```

Generated SQL Code

TableGen syntax in SQLGen

SQL table creation

```
class Table {
  int PrimaryKey = 0;
}

class Customer <string name,
                string affiliation> : Table {
  int ID = PrimaryKey;
  string Name = name;
  string Affiliation = affiliation;
}
```

Input TableGen Code

```
CREATE TABLE Customer (
    ID          int,
    Name        varchar(255),
    Affiliation varchar(255),
    PRIMARY KEY (ID)
);
```

Generated SQL Code

TableGen syntax in SQLGen

SQL table creation

```
class Table {
  int PrimaryKey = 0;
}

class Customer <string name,
                string affiliation> : Table {
  int ID = PrimaryKey;
  string Name = name;
  string Affiliation = affiliation;
}
```

Input TableGen Code

```
CREATE TABLE Customer (
    ID          int,
    Name        varchar(255),
    Affiliation varchar(255),
    PRIMARY KEY (ID)
);
```

Generated SQL Code

TableGen syntax in SQLGen

Inserting rows into a SQL table

Input TableGen Code

Generated SQL Code

TableGen syntax in SQLGen

Inserting rows into a SQL table

```
INSERT INTO Customer (
    ID,
    Name,
    Affiliation
)
VALUES (0, "John Smith", "UC Irvine");
```

Input TableGen Code

Generated SQL Code

TableGen syntax in SQLGen

Inserting rows into a SQL table

```
def john : Customer<"John Smith", "UC Irvine">;
```

Input TableGen Code

```
INSERT INTO Customer (
    ID,
    Name,
    Affiliation
)
VALUES (0, "John Smith", "UC Irvine");
```

Generated SQL Code

TableGen syntax in SQLGen

Inserting rows into a SQL table

```
class Customer <string name,
                string affiliation> : Table {
  int ID = PrimaryKey;
  string Name = name;
  string Affiliation = affiliation;
}

def john : Customer<"John Smith", "UC Irvine">;
```

Input TableGen Code

```
INSERT INTO Customer (
    ID,
    Name,
    Affiliation
)
VALUES (0, "John Smith", "UC Irvine");
```

Generated SQL Code

SQLGen entry point

```
int main(int argc, char **argv) {
  cl::ParseCommandLineOptions(argc, argv);
  return llvm::TableGenMain(argv[0], &CallbackFunc);
}
```

SQLGen entry point

```
int main(int argc, char **argv) {
  cl::ParseCommandLineOptions(argc, argv);
  return llvm::TableGenMain(argv[0], &CallbackFunc);
}
```

```
bool CallbackFunc(raw_ostream &OS, RecordKeeper &Records)
```

SQLGen entry point

```
int main(int argc, char **argv) {
  cl::ParseCommandLineOptions(argc, argv);
  return llvm::TableGenMain(argv[0], &CallbackFunc);
}
```

```
bool CallbackFunc(raw_ostream &OS, RecordKeeper &Records)
```

- **OS**: Stream to the output file (i.e. Output stream to print the SQL code)

SQLGen entry point

```
int main(int argc, char **argv) {
  cl::ParseCommandLineOptions(argc, argv);
  return llvm::TableGenMain(argv[0], &CallbackFunc);
}
```

```
bool CallbackFunc(raw_ostream &OS, RecordKeeper &Records)
```

- **OS**: Stream to the output file (i.e. Output stream to print the SQL code)
- **Records**: In-memory representation of the *parsed* TableGen code

Creating SQL Tables

TableGen syntax in SQLGen

Recap: SQL table creation

```
class Table {
  int PrimaryKey = 0;
}

class Customer <string name,
                string affiliation> : Table {
  int ID = PrimaryKey;
  string Name = name;
  string Affiliation = affiliation;
}
```

Input TableGen Code

```
CREATE TABLE Customer (
    ID          int,
    Name        varchar(255),
    Affiliation varchar(255),
    PRIMARY KEY (ID)
);
```

Generated SQL Code

Enumerating TableGen class-es

```
bool CallbackFunc(raw_ostream &OS, RecordKeeper &Records) {
  const auto &Classes = Records.getClasses();
  ...
}
```

Enumerating TableGen class-es

```
bool CallbackFunc(raw_ostream &OS, RecordKeeper &Records) {
  const auto &Classes = Records.getClasses();
  ...
}
```

```
std::map<                                                    >
```

Enumerating TableGen class-es

```
bool CallbackFunc(raw_ostream &OS, RecordKeeper &Records) {
  const auto &Classes = Records.getClasses();
  ...
}
```

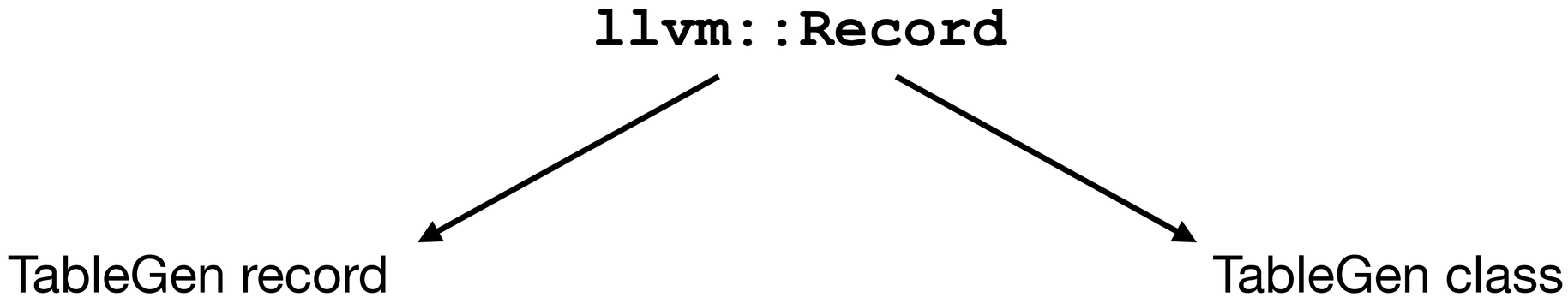
```
std::map<std::string,                                    >
```

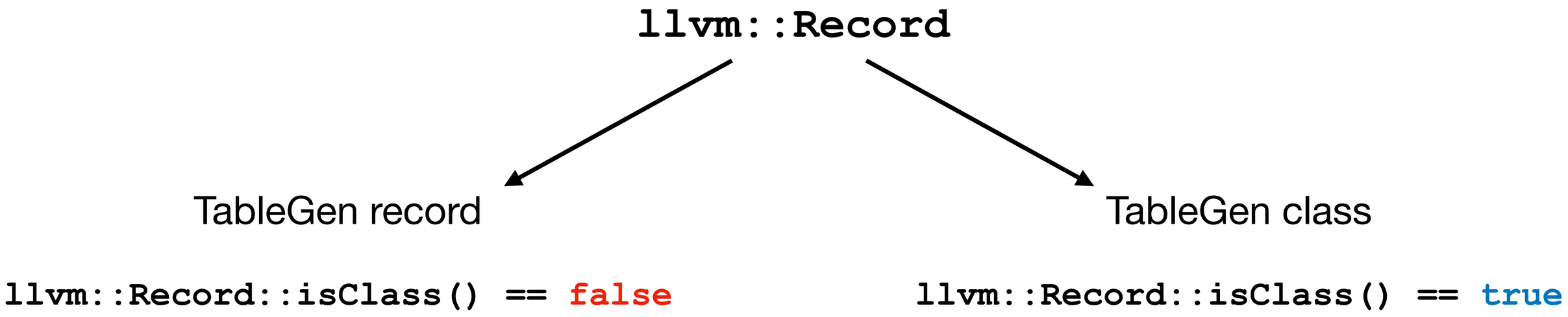
Enumerating TableGen class-es

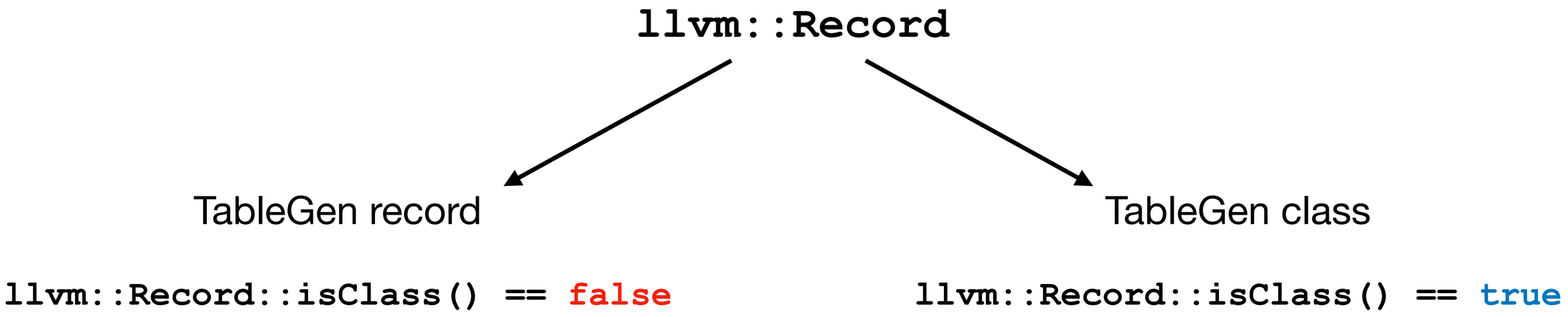
```
bool CallbackFunc(raw_ostream &OS, RecordKeeper &Records) {
  const auto &Classes = Records.getClasses();
  ...
}
```

```
std::map<std::string, std::unique_ptr<llvm::Record>>
```

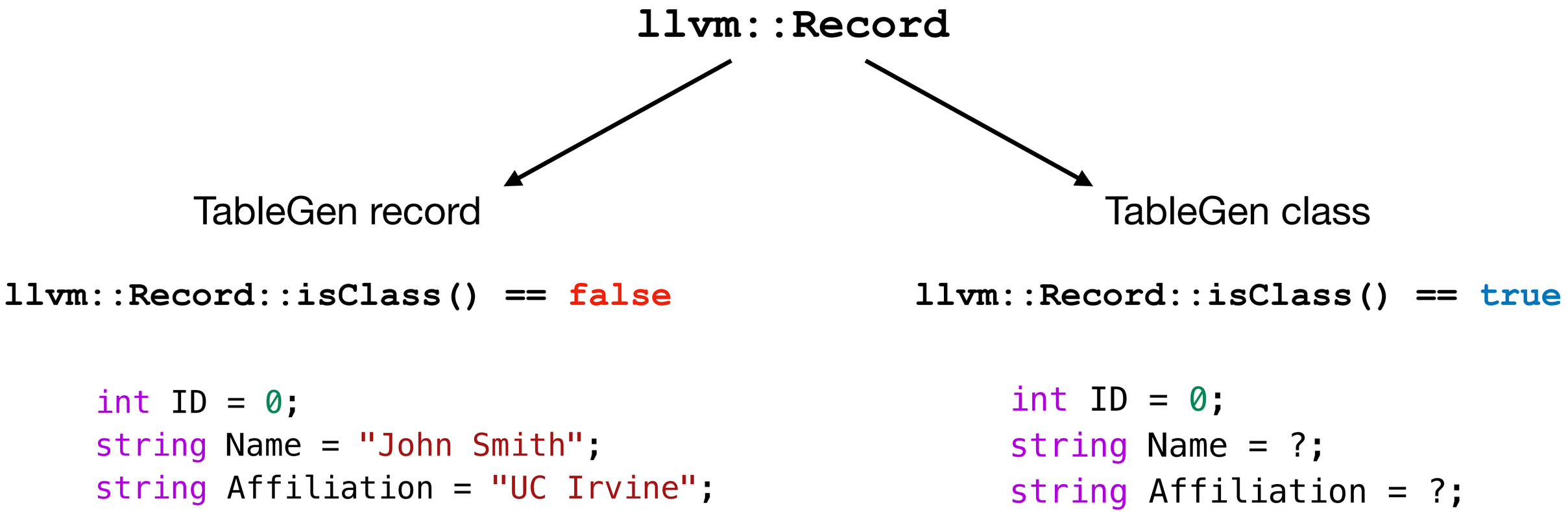
`llvm::Record`


llvm::Record
TableGen record
TableGen class


llvm::Record
TableGen record
TableGen class
llvm::Record::isClass() == false
llvm::Record::isClass() == true




```
int ID = 0;
string Name = "John Smith";
string Affiliation = "UC Irvine";
```


llvm::Record
TableGen record
TableGen class
llvm::Record::isClass() == false
llvm::Record::isClass() == true
int ID = 0;
string Name = "John Smith";
string Affiliation = "UC Irvine";
int ID = 0;
string Name = ?;
string Affiliation = ?;

```
class Customer <string name, string affiliation> : Table {
  int ID = PrimaryKey;
  string Name = name;
  string Affiliation = affiliation;
}


def john : Customer<"John Smith", "UC Irvine">;
```

```
class Customer <string name, string affiliation> : Table {
  int ID = PrimaryKey;
  string Name = name;
  string Affiliation = affiliation;
}


def john {
  int PrimaryKey = 0;
  int ID = 0;
  string Name = "John Smith";
  string Affiliation = "UC Irvine";
}
```

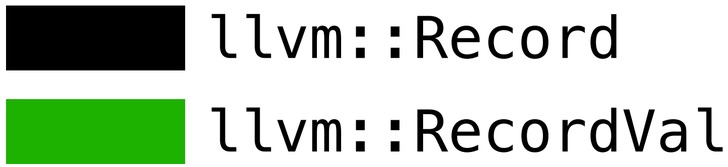
```
class Customer <string name, string affiliation> : Table {
  int ID = PrimaryKey;
  string Name = name;
  string Affiliation = affiliation;
}
```

```
def john {
  int PrimaryKey = 0;
  int ID = 0;
  string Name = "John Smith";
  string Affiliation = "UC Irvine";
}
```

[redacted] llvm::Record

```
class Customer <string name, string affiliation> : Table {
  int ID = PrimaryKey;
  string Name = name;
  string Affiliation = affiliation;
}
```

```
def john {
  int PrimaryKey = 0;
  int ID = 0;
  string Name = "John Smith";
  string Affiliation = "UC Irvine";
}
```



```
const auto &Classes = Records.getClasses();
for (const auto &P : Classes) {
  auto ClassName = P.first;
  Record &ClassRecord = *P.second;









}
```

```
const auto &Classes = Records.getClasses();
for (const auto &P : Classes) {
  auto ClassName = P.first;
  Record &ClassRecord = *P.second;
  OS << "CREATE TABLE " << ClassName << " (";




  OS << ");\n";
}
```

```
CREATE TABLE Customer (


);
```

```
const auto &Classes = Records.getClasses();
for (const auto &P : Classes) {
  auto ClassName = P.first;
  Record &ClassRecord = *P.second;
  OS << "CREATE TABLE " << ClassName << " (";
  for (const RecordVal &RV : ClassRecord.getValues()) {




  }
  OS << ");\n";
}
```

```
CREATE TABLE Customer (


);
```

```
const auto &Classes = Records.getClasses();
for (const auto &P : Classes) {
  auto ClassName = P.first;
  Record &ClassRecord = *P.second;
  OS << "CREATE TABLE " << ClassName << " (";
  for (const RecordVal &RV : ClassRecord.getValues()) {
    OS << "\t" << RV.getName() << " ";




  }
  OS << ");\n";
}
```

```
CREATE TABLE Customer (
    ID
    Name
    Affiliation

);
```

```
const auto &Classes = Records.getClasses();
for (const auto &P : Classes) {
  auto ClassName = P.first;
  Record &ClassRecord = *P.second;
  OS << "CREATE TABLE " << ClassName << " (";
  for (const RecordVal &RV : ClassRecord.getValues()) {
    OS << "\t" << RV.getName() << " ";
    if (isa<IntRecTy>(RV.getType()))
      OS << "int,";


  }
  OS << ");\n";
}
```

```
CREATE TABLE Customer (
    ID          int,
    Name
    Affiliation

);
```

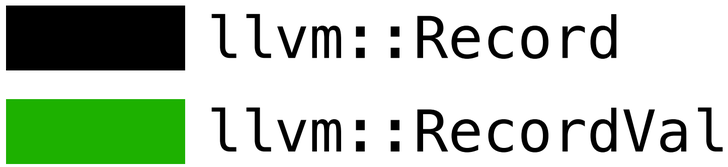
```
const auto &Classes = Records.getClasses();
for (const auto &P : Classes) {
  auto ClassName = P.first;
  Record &ClassRecord = *P.second;
  OS << "CREATE TABLE " << ClassName << " (";
  for (const RecordVal &RV : ClassRecord.getValues()) {
    OS << "\t" << RV.getName() << " ";
    if (isa<IntRecTy>(RV.getType()))
      OS << "int,";
    if (isa<StringRecTy>(RV.getType()))
      OS << "varchar(255),";
  }
  OS << ");\n";
}
```

```
CREATE TABLE Customer (
    ID          int,
    Name        varchar(255),
    Affiliation varchar(255),

);
```

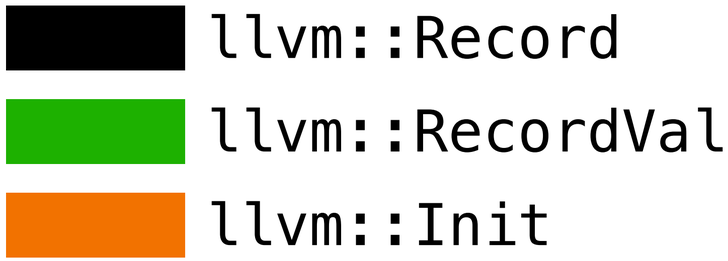
```
class Customer <string name, string affiliation> : Table {
  int ID = PrimaryKey;
  string Name = name;
  string Affiliation = affiliation;
}
```

```
def john {
  int PrimaryKey = 0;
  int ID = 0;
  string Name = "John Smith";
  string Affiliation = "UC Irvine";
}
```

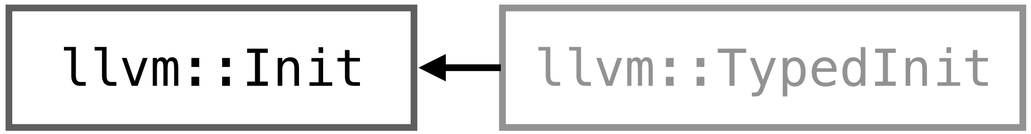


```
class Customer <string name, string affiliation> : Table {
  int ID = PrimaryKey;
  string Name = name;
  string Affiliation = affiliation;
}
```

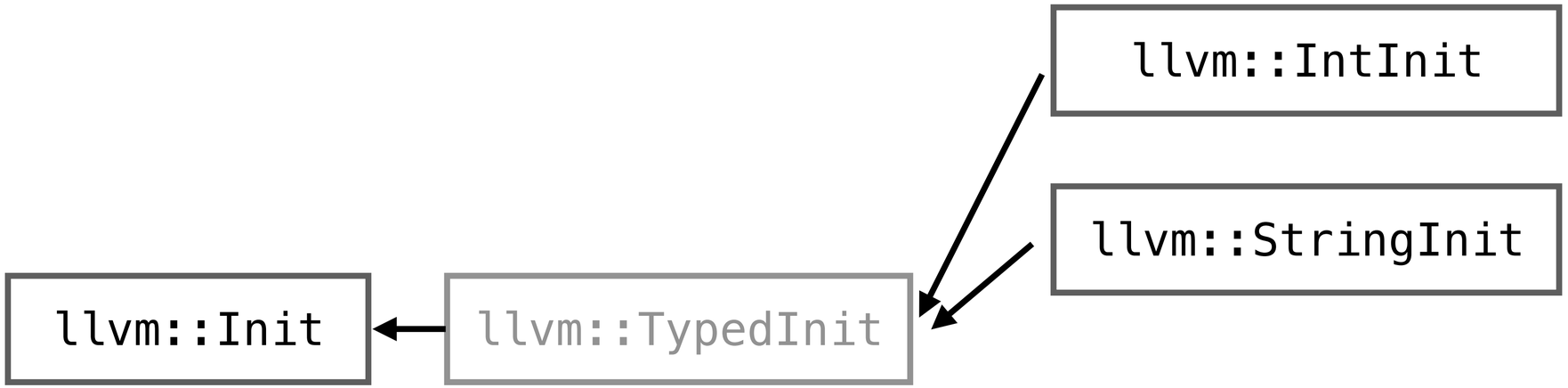
```
def john {
  int PrimaryKey = 0;
  int ID = 0;
  string Name = "John Smith";
  string Affiliation = "UC Irvine";
}
```



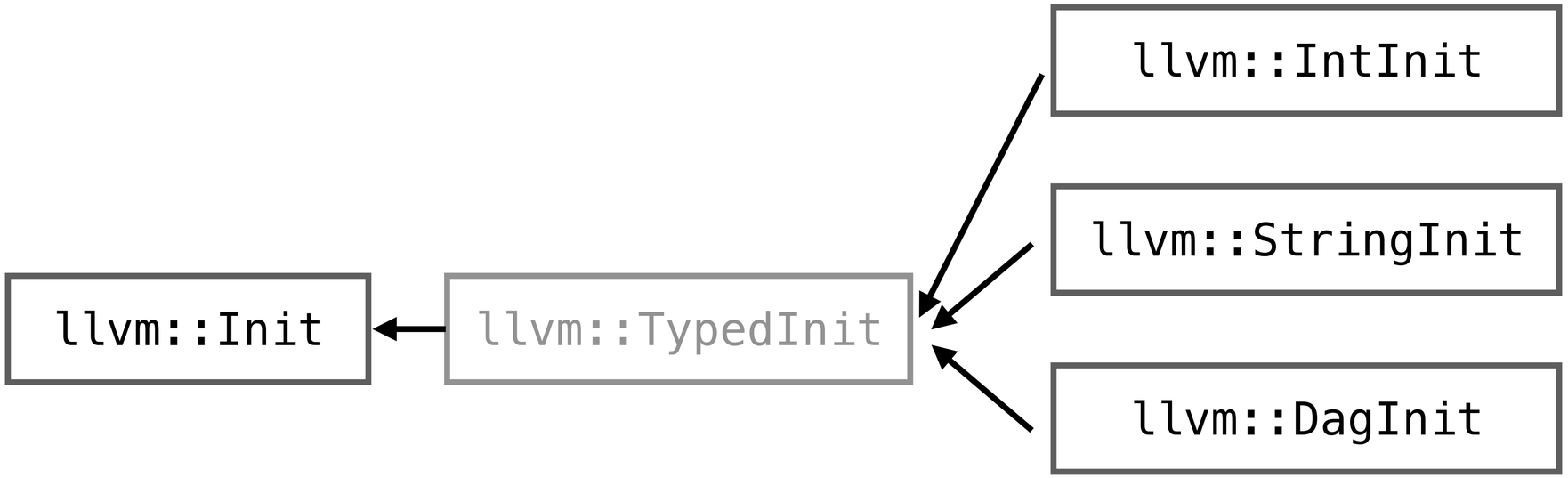
Common derived classes of llvm::Init



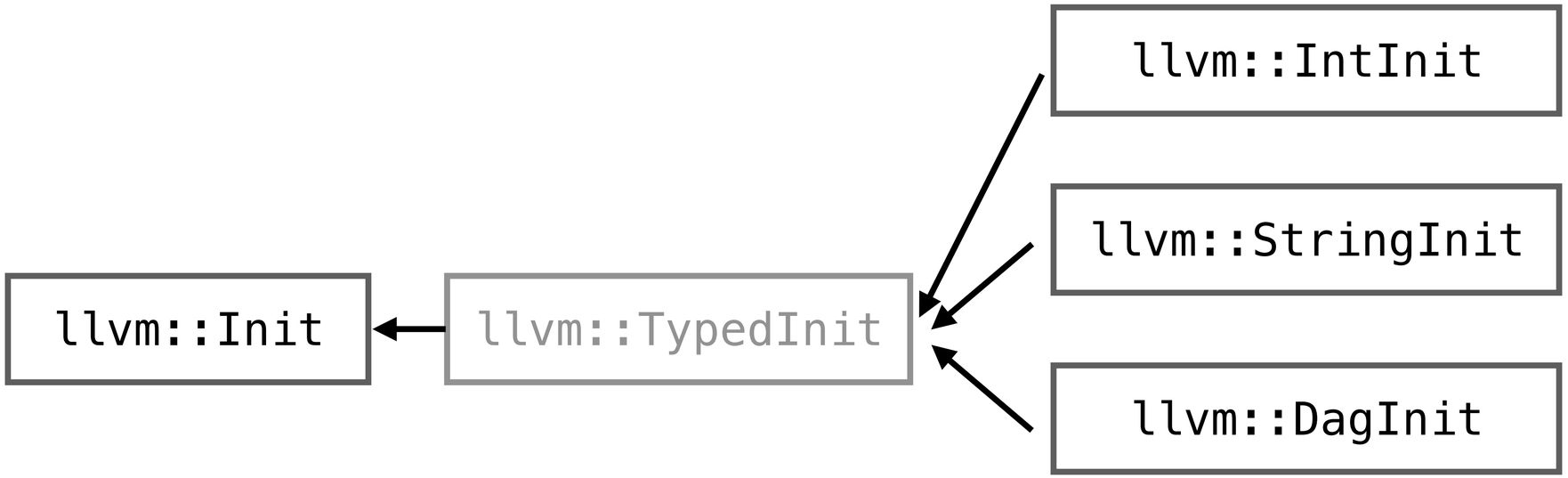
Common derived classes of llvm::Init



Common derived classes of llvm::Init

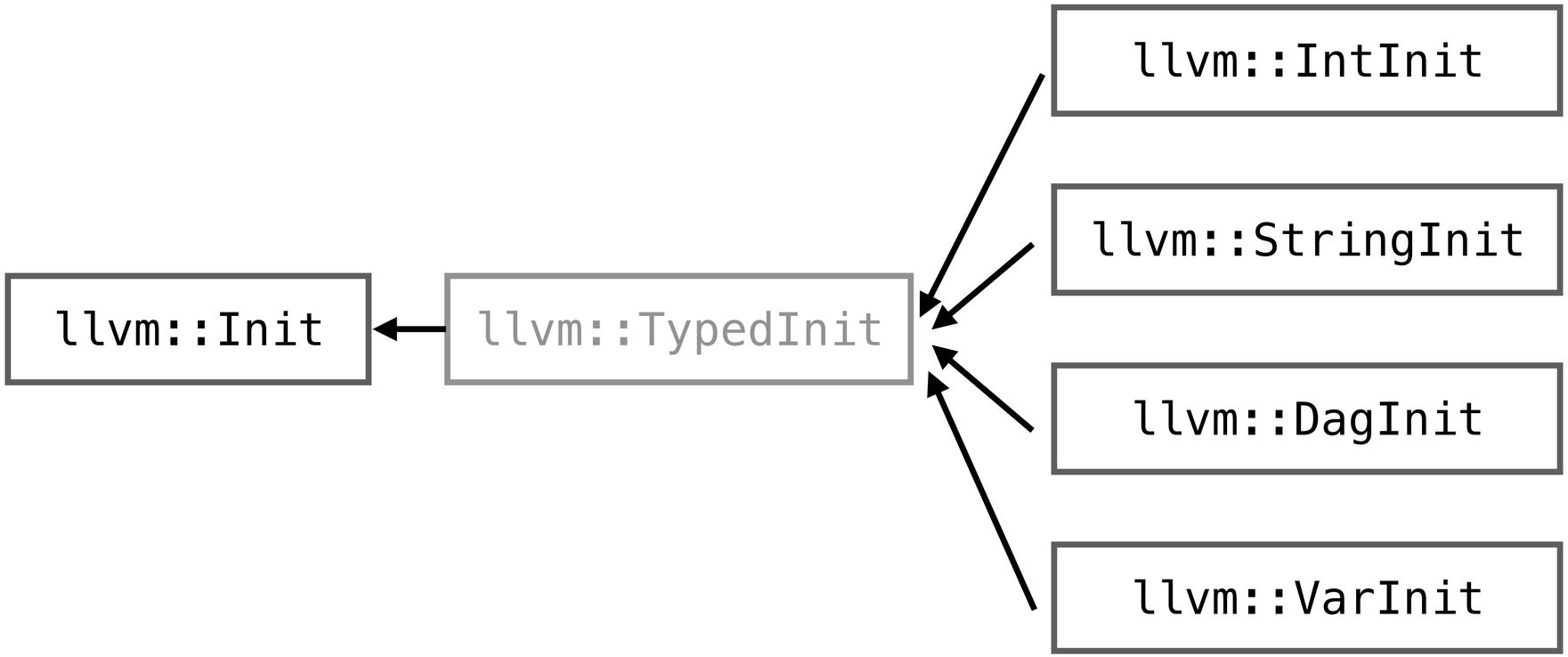


Common derived classes of llvm::Init




```
int PrimaryKey = 0;
int ID = PrimaryKey;
```

Common derived classes of llvm::Init




```
int PrimaryKey = 0;
int ID = PrimaryKey;
```

```
for (const auto &P : Classes) {
  ...
  OS << "CREATE TABLE " << ClassName << " (";
  for (const RecordVal &RV : ClassRecord.getValues()) {
    OS << "\t" << RV.getName() << " ";
    ...

  }
  OS << ");\n";
}
```

```
CREATE TABLE Customer (
    ID          int,
    Name        varchar(255),
    Affiliation varchar(255),

);
```

```
for (const auto &P : Classes) {
  ...
  OS << "CREATE TABLE " << ClassName << " (";
  for (const RecordVal &RV : ClassRecord.getValues()) {
    OS << "\t" << RV.getName() << " ";
    ...
    Init *Val = RV.getValue();




  }
  OS << ");\n";
}
```

```
CREATE TABLE Customer (
    ID          int,
    Name        varchar(255),
    Affiliation varchar(255),

);
```

```
for (const auto &P : Classes) {
  ...
  OS << "CREATE TABLE " << ClassName << " (";
  for (const RecordVal &RV : ClassRecord.getValues()) {
    OS << "\t" << RV.getName() << " ";
    ...
    Init *Val = RV.getValue();
    if (auto *VI = dyn_cast<VarInit>(Val)) {
      if (VI->getName() == "PrimaryKey")
        OS << "PRIMARY KEY (" << RV.getName() << ")";
    }
  }
  OS << ");\n";
}
```

```
CREATE TABLE Customer (
    ID          int,
    Name        varchar(255),
    Affiliation varchar(255),
    PRIMARY KEY (ID)
);
```

SQL Row Insertion

TableGen syntax in SQLGen

Recap: Inserting rows into a SQL table

```
class Customer <string name,
                string affiliation> : Table {
  int ID = PrimaryKey;
  string Name = name;
  string Affiliation = affiliation;
}

def john : Customer<"John Smith", "UC Irvine">;
```

Input TableGen Code

```
INSERT INTO Customer (
    ID,
    Name,
    Affiliation
)
VALUES (0, "John Smith", "UC Irvine");
```

Generated SQL Code

Enumerating TableGen Records

```
bool CallbackFunc(raw_ostream &OS, RecordKeeper &Records) {



}
```

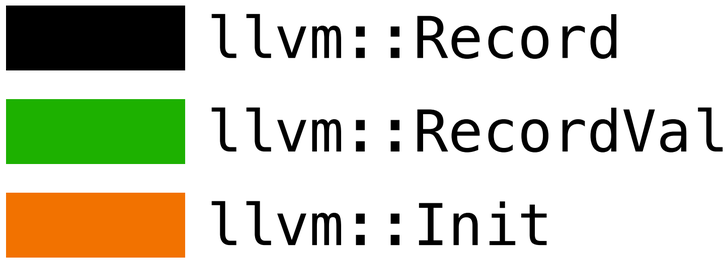
Enumerating TableGen Records

```
bool CallbackFunc(raw_ostream &OS, RecordKeeper &Records) {
  auto SQLRows = Records.getAllDerivedDefinitions("Table");
  for (const Record *RowRecord : SQLRows) {
    ...
  }
}
```

Recap: In-memory representations for TableGen records / classes

```
class Customer <string name, string affiliation> : Table {
  int ID = PrimaryKey;
  string Name = name;
  string Affiliation = affiliation;
}
```

```
def john {
  int PrimaryKey = 0;
  int ID = 0;
  string Name = "John Smith";
  string Affiliation = "UC Irvine";
}
```



```
for (const Record *RowRecord : SQLRows) {
  for (const RecordVal &RV : RowRecord->getValues()) {

  }




}
```

```
                                                    INSERT INTO Customer (

for (const Record *RowRecord : SQLRows) {                        )
  OS << "INSERT INTO " << ClassName << " (\n";
  for (const RecordVal &RV : RowRecord->getValues()) {


  }
  OS << ")\n";




}
```

```
INSERT INTO Customer (
    ID,
    Name,
    Affiliation
)
```

```
for (const Record *RowRecord : SQLRows) {
  OS << "INSERT INTO " << ClassName << " (\n";
  for (const RecordVal &RV : RowRecord->getValues()) {
    auto Name = RV.getName();
    OS << "\t" << Name << ",\n";
  }
  OS << ")\n";









}
```

```
for (const Record *RowRecord : SQLRows) {
  OS << "INSERT INTO " << ClassName << " (\n";
  for (const RecordVal &RV : RowRecord->getValues()) {
    auto Name = RV.getName();
    OS << "\t" << Name << ",\n";
  }
  OS << ")\n";
  OS << "VALUES (";
  for (const RecordVal &RV : RowRecord->getValues()) {



  }
  OS << ");\n";
}
```

```
INSERT INTO Customer (
    ID,
    Name,
    Affiliation
)
VALUES (                                    );
```

```
for (const Record *RowRecord : SQLRows) {
  OS << "INSERT INTO " << ClassName << " (\n";
  for (const RecordVal &RV : RowRecord->getValues()) {
    auto Name = RV.getName();
    OS << "\t" << Name << ",\n";
  }
  OS << ")\n";
  OS << "VALUES (";
  for (const RecordVal &RV : RowRecord->getValues()) {
    const Init *Val = RV.getValue();
    OS << Val->getAsString() << ", ";
  }
  OS << ");\n";
}
```

```
INSERT INTO Customer (
    ID,
    Name,
    Affiliation
)
VALUES (0, "John Smith", "UC Irvine");
```

Making SQL Queries

TableGen syntax in SQLGen

Making queries

```
class Query <string table,
             dag query_fields, dag condition> {



}
```

Input TableGen Code

```
SELECT                  FROM
```

Generated SQL Code

TableGen syntax in SQLGen

Making queries

```
class Query <string table,
             dag query_fields, dag condition> {

  dag Fields = query_fields;

}
```

Input TableGen Code

```
SELECT Affiliation FROM
```

Generated SQL Code

TableGen syntax in SQLGen

Making queries

```
class Query <string table,
             dag query_fields, dag condition> {
  string TableName = table;
  dag Fields = query_fields;

}
```

Input TableGen Code

```
SELECT Affiliation FROM Customer
```

Generated SQL Code

TableGen syntax in SQLGen

Making queries

```
class Query <string table,
             dag query_fields, dag condition> {
  string TableName = table;
  dag Fields = query_fields;
  dag WhereClause = condition;
}
```

Input TableGen Code

```
SELECT Affiliation FROM Customer
WHERE Name = "John Smith";
```

Generated SQL Code

TableGen syntax in SQLGen

Making queries

```
class Query <string table,
             dag query_fields, dag condition> {
  string TableName = table;
  dag Fields = query_fields;
  dag WhereClause = condition;
}

def : Query<"Customer", (fields "Affiliation"),
            (eq "Name", "John Smith"     )>;
```

Input TableGen Code

```
SELECT Affiliation FROM Customer
WHERE Name = "John Smith";
```

Generated SQL Code

TableGen syntax in SQLGen

Making queries

```
class Query <string table,
             dag query_fields, dag condition> {
  string TableName = table;
  dag Fields = query_fields;
  dag WhereClause = condition;
}

def : Query<"Customer", (fields "Affiliation"),
            (eq "Name", "John Smith"     )>;
```

Anonymous record

Input TableGen Code

```
SELECT Affiliation FROM Customer
WHERE Name = "John Smith";
```

Generated SQL Code

TableGen syntax in SQLGen

Making queries

```
class Query <string table,
             dag query_fields, dag condition> {
  string TableName = table;
  dag Fields = query_fields;
  dag WhereClause = condition;
}

def : Query<"Customer", (fields "Affiliation"),
            (eq "Name", "John Smith"     )>;
```

Anonymous record

Input TableGen Code

```
SELECT Affiliation FROM Customer
WHERE Name = "John Smith";
```

Generated SQL Code

Example of the dag type

An expression tree

Modeling expression: *9 + 4 * (x - 3)*

```
def plus;
def minus;



dag expr = (                                        );
```

Example of the dag type

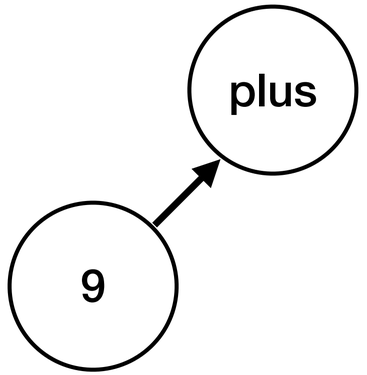
An expression tree

Modeling expression: *9 + 4 * (x - 3)*

```
def plus;
def minus;



dag expr = (plus 9,                                    );
```



Example of the dag type

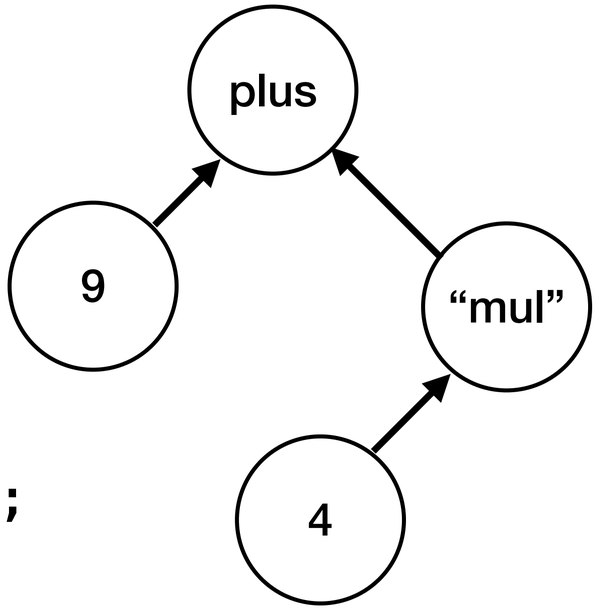
An expression tree

Modeling expression: *9 + 4 * (x - 3)*

```
def plus;
def minus;



dag expr = (plus 9, ("mul" 4,                    ));
```



Example of the dag type

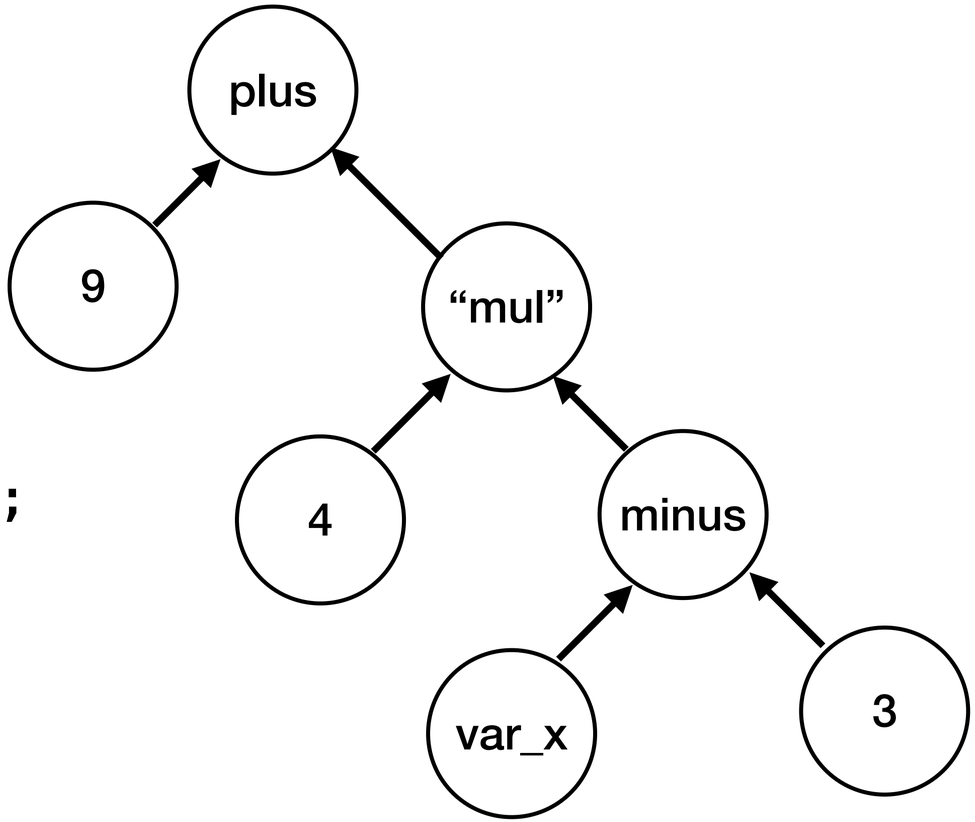
An expression tree

Modeling expression: *9 + 4 * (x - 3)*

```
def plus;
def minus;

def var_x : Var {...}

dag expr = (plus 9, ("mul" 4, (minus var_x, 3)));
```



Making SQL queries

More examples

TableGen

```
def : Query<"Orders", (fields "Person", "Amount"),
            (gt "Amount", 8)>
```

SQL

```
SELECT Person, Amount FROM Orders
WHERE Amount > 8;
```

Making SQL queries

More examples

TableGen

```
def : Query<"Orders", (fields "Person", "Amount"),
            (gt "Amount", 8)>
```

SQL

```
SELECT Person, Amount FROM Orders
WHERE Amount > 8;
```

TableGen

```
def : Query<"Orders", (fields "ProductName", "Person"),
            (and (gt "Amount", 8), (ne "Person", 1))>;
```

SQL

```
SELECT ProductName, Person FROM Orders
WHERE Amount > 8 AND Person <> 1;
```

TableGen syntax in SQLGen

Making queries

```
class Query <string table,
             dag query_fields, dag condition> {
  string TableName = table;
  dag Fields = query_fields;
  dag WhereClause = condition;
}

def : Query<"Customer", (fields "Affiliation"),
            (eq "Name", "John Smith"     )>;
```

Input TableGen Code

```
SELECT Affiliation FROM Customer
WHERE Name = "John Smith";
```

Generated SQL Code

TableGen syntax in SQLGen

Making queries

```
class Query <string table,
             dag query_fields, dag condition> {
  string TableName = table;
  dag Fields = query_fields;
  dag WhereClause = condition;
}

def : Query<"Customer", (fields "Affiliation"),
            (eq "Name", "John Smith":$str)>;
```

Input TableGen Code

```
SELECT Affiliation FROM Customer
WHERE Name = "John Smith";
```

Generated SQL Code

TableGen syntax in SQLGen

Making queries

```
class Query <string table,
             dag query_fields, dag condition> {
  string TableName = table;
  dag Fields = query_fields;
  dag WhereClause = condition;
}

def : Query<"Customer", (fields "Affiliation"),
            (eq "Name", "John Smith":$str)>;
```

An argument with *tag*

Input TableGen Code

```
SELECT Affiliation FROM Customer
WHERE Name = "John Smith";
```

Generated SQL Code

```
auto SQLQueries = Records.getAllDerivedDefinitions("Query");
for (const Record *Query : SQLQueries) {









}
```

```
auto SQLQueries = Records.getAllDerivedDefinitions("Query");
for (const Record *Query : SQLQueries) {
  auto TableName = Query->getValueAsString("TableName");
  const DagInit *Fields = Query->getValueAsDag("Fields");




}
```

SELECT

```
auto SQLQueries = Records.getAllDerivedDefinitions("Query");
for (const Record *Query : SQLQueries) {
  auto TableName = Query->getValueAsString("TableName");
  const DagInit *Fields = Query->getValueAsDag("Fields");
  OS << "SELECT ";



}
```

```
                                                    SELECT Affiliation
```

```
auto SQLQueries = Records.getAllDerivedDefinitions("Query");
for (const Record *Query : SQLQueries) {
  auto TableName = Query->getValueAsString("TableName");
  const DagInit *Fields = Query->getValueAsDag("Fields");
  OS << "SELECT ";
  for (const Init *Arg : Fields->getArgs())
    OS << Arg->getAsUnquotedString() << ",";

}
```

```
SELECT Affiliation FROM Customer
```

```
auto SQLQueries = Records.getAllDerivedDefinitions("Query");
for (const Record *Query : SQLQueries) {
  auto TableName = Query->getValueAsString("TableName");
  const DagInit *Fields = Query->getValueAsDag("Fields");
  OS << "SELECT ";
  for (const Init *Arg : Fields->getArgs())
    OS << Arg->getAsUnquotedString() << ",";
  OS << " FROM " << TableName << "\n";
}
```

```
void visitWhereClause(const DagInit *Term, raw_ostream &OS) {
  const Init *Operator = Term->getOperator();

}
```

TableGen

```
(                                    )
```

SQL

```
SELECT Affiliation FROM Customer
WHERE
```

```
void visitWhereClause(const DagInit *Term, raw_ostream &OS) {
  const Init *Operator = Term->getOperator();
```

TableGen

```
(eq                                    )
```

SQL

```
SELECT Affiliation FROM Customer
WHERE
```

```
}
```

```
void visitWhereClause(const DagInit *Term, raw_ostream &OS) {
  const Init *Operator = Term->getOperator();
  for (int i = 0; i < Term->arg_size(); ++i) {
    const Init *Arg = Term->getArg(i);









  }
}
```

TableGen

```
(eq                                    )
```

SQL

```
SELECT Affiliation FROM Customer
WHERE
```

```
void visitWhereClause(const DagInit *Term, raw_ostream &OS) {
  const Init *Operator = Term->getOperator();
  for (int i = 0; i < Term->arg_size(); ++i) {
    const Init *Arg = Term->getArg(i);
    if (const auto *ArgDag = dyn_cast<DagInit>(Arg))
      visitWhereClause(ArgDag, OS);
```

TableGen

```
(eq                                    )
```

SQL

```
SELECT Affiliation FROM Customer
WHERE
```

```
  }
}
```

```
void visitWhereClause(const DagInit *Term, raw_ostream &OS) {
  const Init *Operator = Term->getOperator();
  for (int i = 0; i < Term->arg_size(); ++i) {
    const Init *Arg = Term->getArg(i);
    if (const auto *ArgDag = dyn_cast<DagInit>(Arg))
      visitWhereClause(ArgDag, OS);
    else {



    }


  }
}
```

TableGen

```
(eq                                    )
```

SQL

```
SELECT Affiliation FROM Customer
WHERE
```

```
void visitWhereClause(const DagInit *Term, raw_ostream &OS) {
  const Init *Operator = Term->getOperator();
  for (int i = 0; i < Term->arg_size(); ++i) {
    const Init *Arg = Term->getArg(i);
    if (const auto *ArgDag = dyn_cast<DagInit>(Arg))
      visitWhereClause(ArgDag, OS);
    else {
      if (Term->getArgName(i) == "str")
        OS << Arg->getAsString();



    }



  }
}
```

TableGen

```
(eq          "John Smith":$str)
```

SQL

```
SELECT Affiliation FROM Customer
WHERE         “John Smith";
```

```
void visitWhereClause(const DagInit *Term, raw_ostream &OS) {
  const Init *Operator = Term->getOperator();
  for (int i = 0; i < Term->arg_size(); ++i) {
    const Init *Arg = Term->getArg(i);
    if (const auto *ArgDag = dyn_cast<DagInit>(Arg))
      visitWhereClause(ArgDag, OS);
    else {
      if (Term->getArgName(i) == "str")
        OS << Arg->getAsString();
      else
        OS << Arg->getAsUnquotedString();
    }


  }
}
```

TableGen

```
(eq "Name", "John Smith":$str)
```

SQL

```
SELECT Affiliation FROM Customer
WHERE Name   “John Smith";
```

```
void visitWhereClause(const DagInit *Term, raw_ostream &OS) {
  const Init *Operator = Term->getOperator();
  for (int i = 0; i < Term->arg_size(); ++i) {
    const Init *Arg = Term->getArg(i);
    if (const auto *ArgDag = dyn_cast<DagInit>(Arg))
      visitWhereClause(ArgDag, OS);
    else {
      if (Term->getArgName(i) == "str")
        OS << Arg->getAsString();
      else
        OS << Arg->getAsUnquotedString();
    }
    if (i < Term->arg_size() - 1)
      printOperator(Operator);
  }
}
```

TableGen

```
(eq "Name", "John Smith":$str)
```

SQL

```
SELECT Affiliation FROM Customer
WHERE Name = "John Smith";
```

Epilogue

Other useful TableGen syntax

The `multiclass` - creating multiple records at once

TableGen operators (a.k.a *bang* operators) - e.g. `!add`, `!mul`, `!or`

Bits slice

String concatenation via ‘#’

Casting from string to a record

Sample code

https://github.com/mshockwave/SQLGen

Some additional features / highlights…

- Hierarchical records via `FOREIGN KEY`
- Ordering fields via `ORDER BY`
- Using LLVM LIT for testing

Q&A

GitHub: mshockwave

Email: minyihh@uci.edu

Book URL: https://tinyurl.com/3xnc5r3t