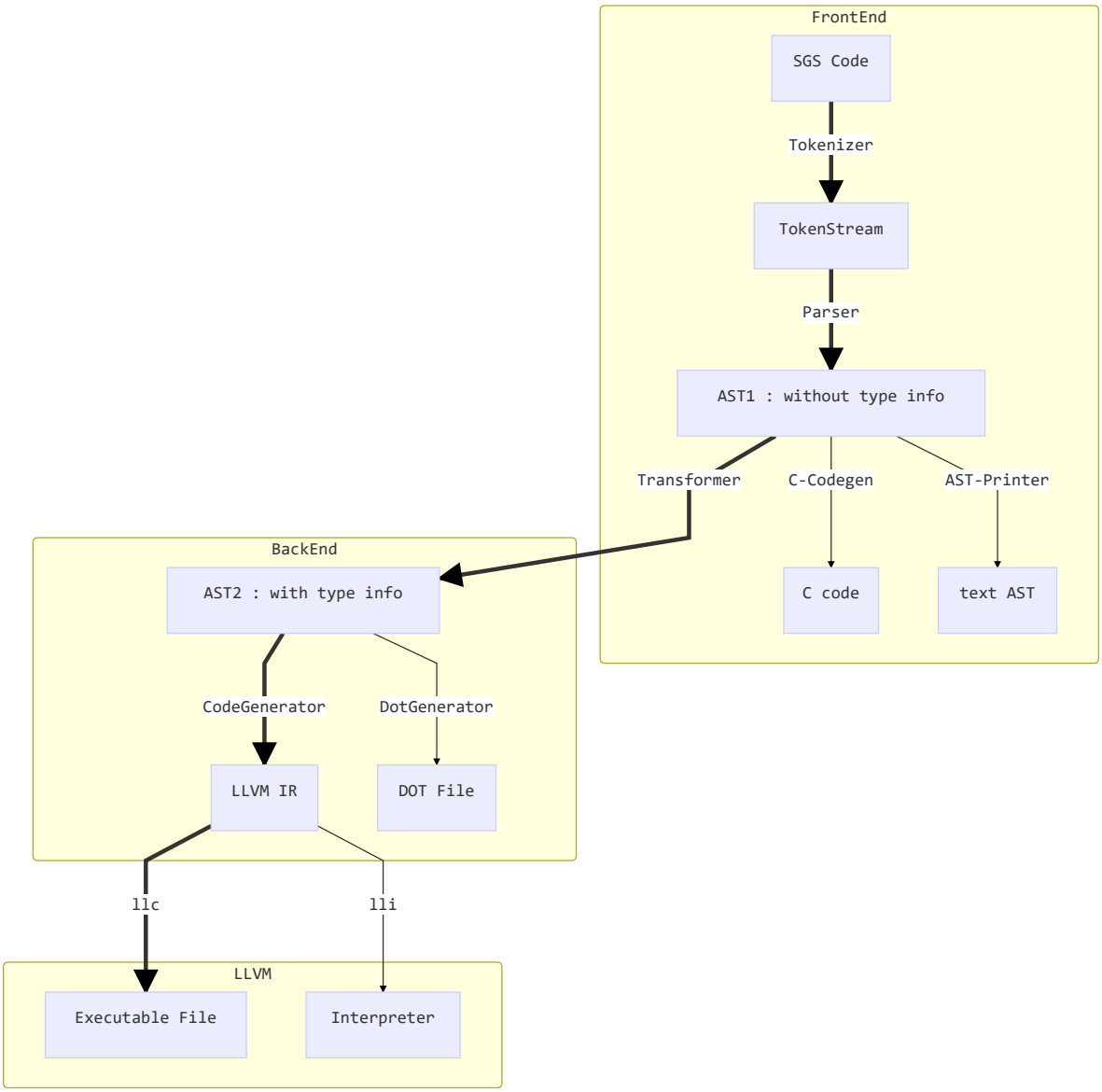
SGS 编译器文档

前言

项目概览

项目架构如下图所示。编译器前端读取代码，对代码进行词法分析、语法分析，提供不含有类型标注的抽象语法树给编译器后端。后端对抽象语法树添加类型标注并检查语法树结构，将语法树生成为可正确执行的 LLVM IR。编译器前后端之间高内聚低耦合，后端可以适配多种类 C 语法的语言的代码生成。以 LLVM IR 为目标语言可以借助 LLVM 这一平台轻松的实现多种汇编代码的生成以及平台无关/有关的代码优化，同时也可以与其它语言在 LLVM IR 层面上实现相互调用或者 FFI ，从而使语言本身获得更加强大的适配性。



编译器命令行接口

编译器可执行文件名字默认为 `sgs.exe` 。将其添加到 PATH 后可以在命令行中调用其功能。一些扩展功能需要用户环境中包含相应的工具。下面是它的命令行帮助界面：

```
λ sgs -help
USAGE: sgs [options] <input file>

OPTIONS:

General options:

```

```
  -emit-ast            - Print abstract syntax tree in console
  -emit-cpp            - To generate cpp file
  -emit-dot            - To generate dot file
  -emit-ir             - To generate LLVM IR
  -emit-png            - To generate png file by dot
  -i                   - Just-in-time execute with lli
  -o=<output filename> - Generate executable file with llvm and clang facilities

Generic Options:

  -help                - Display available options (-help-hidden for more)
  -help-list           - Display list of available options (-help-list-hidden for more)
  -version             - Display the version of this program
```

依赖的 LLVM 工具包括 `clang` , `lli` , `llvm-as` , `llc`

SGS 语言手册

SGS 语言简介

SGS语言为一款轻量级程序设计语言，具有很强的可移植性和可扩展性。目前开发团队已经完成 SGS 编译器及解释器，使其可以覆盖各类应用场景的需求。

在编译器前端，SGS 侧重编程语言与自然语言的相似度，使得 SGS 代码更接近于英语，从而可以让绝大多数人们可以理解其内部含义。在编译器后端，SGS 侧重对象的处理与类型系统的维护。在 SGS 中，非基本类型的变量全部作为对象处理。这样的好处是避免了指针的使用，从而让 SGS 代码更加安全。

SGS 语言语法

SGS语法与英语有很大相似度。

- 变量声明与赋值语句：`let <type> <var> [be <exp>].`

  其中 `let` 和 `be` 为关键字（ `be` 及其后 `<exp>` 可省）， `<type>`为变量类型， `<var>` 为变量名字。若变量已声明过，则省略type。

- 函数声明语句：`new function <name> [with <type1> <para1>, <type2> <para 2>...] return <type>.`

其中 `new`、`function`、`with` 和 `return` 为关键字，`name` 为函数名，`<type1>`、`<para1>` 及其后的 `<type>` 和 `<para>` 为参数的类型和名字，若无参数，则省略 `with` 及其与 `return` 之间的内容。`<type>` 为函数返回值，若无返回值，则省略 `return` 及其后内容。

- 函数定义语句 `start <name>. <block>. end <name>.`

  其中 `start` 和 `end` 为关键字，`<name>` 为函数名。在 `<block>` 中定义函数主体，SGS 默认为每个有返回值的函数定义一个名为 `result` 的局部变量。

- 类声明语句 `new class <name> with <type1> <para1>[, <type2> <para2>...]`

  其中 `new`、`class` 和 `with` 为关键字，`<name>` 为类名，`type1`、`para1` 及其后的 `type2` 和 `para2` 为内部成员变量的类型和名字。

- 函数调用语句 `<func> with <para1> <exp1>, <para2> <exp2> ...`，`<func> with <exp1>, <exp2>...`

  其中 `with` 为关键字，`<func>` 为函数名。SGS 函数调用支持两种方式：显式参数传递与隐式参数传递。第一种为显式参数传递，`<para1>` 为形参，`<exp1>` 为实参，此种传参方式支持乱序，即参数顺序不必与声明顺序一致。第二种为隐式参数传递，`<exp1>` 为实参，此种方式与 C 类似，参数顺序必须与声明顺序一致。

- 分支语句 `if <exp> then <block>[ else <block>] end if.`

  其中`if`、`then`、`else` 和 `end` 为关键字，`<exp>` 为判断条件，`<block>` 为语句块。如果判断条件为真的话，执行 `then` 后的 `<block>` 。

- 循环语句 `loop when <exp> <block> end loop.`

  其中 `loop`、`when` 和 `end` 为关键字，`<exp>` 为判断条件，`<block>` 为语句块。当判断条件为真时循环执行语句块，否则跳出循环。

- 跳转语句 `return | break | continue`

  SGS中一共三种直接跳转语句，`return` 为跳出函数体，`break` 为跳出循环体，`continue` 为重做循环体。

SGS 语言语义

编译器设计

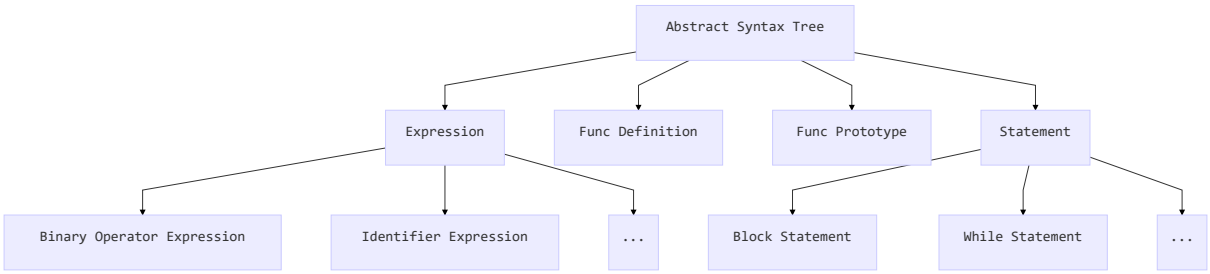
因为 SGS 编译器的目标代码是 LLVM IR，所以我们把从代码到抽象语法树的工作分为编译器的前端，而把对抽象语法树进行语义分析以及生成目标代码作为编译器的后端。

通用部分

为了防止编译器前后端工作因为阻塞而导致无意义的时间浪费，决定通过约定好抽象语法树的结构来使前后端工作同时开展。其中作为编译器前后端的桥梁的就是抽象语法树。通过交流我们确定了编译器前端生成的抽象语法树类型 AST1

AST1

在代码中， AST1 对应的是 `sgs::AST` 及其派生类组成的一组类型，具体的实现在 "SGS/syntax.h" 中

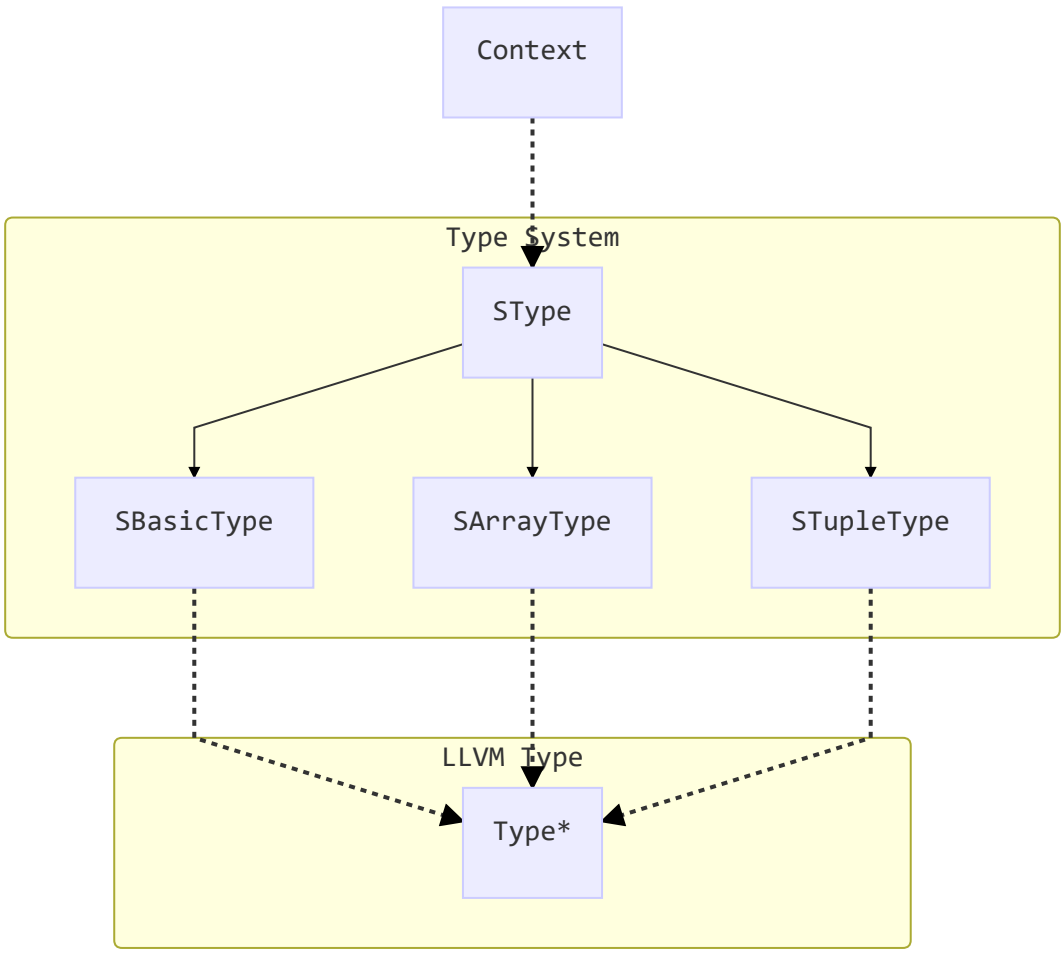


AST 的结构和语言的定义是一一对应的，语言中的所有结构都会对应于抽象语法树中的一部分。AST1 保存了代码中的大部分有意义的信息，但没有根据代码**分析**出其它的信息，它的作用仅仅是作为一个序列化的代码片段传递到编译器后端。生成的 AST 可能包含许多错误的信息，例如在循环体外的 `break` 或 `continue` 等等，这些错误将在后面的语义分析中找到 。

类型系统实现与 AST2

AST2 是 SGS 编译器后端中主要使用的抽象语法树结构。在编译器前端将代码序列化之后生成的 AST1 会通过一次语义分析转化为 AST2， 并为抽象语法树中所有的表达式添加上类型标注、检查函数（包括内置函数以及二元/一元运算符）的类型匹配情况，检查表达式中是否会发生隐式的类型转换等等。

AST2 和 SGS 编译器后端使用的类型系统类图如下：



类似于 `LLVMContext` 的设计思路，这里所有的类型都由 `Context` 统一进行资源管理，并隐藏 `SType` 的构造函数使用户无法通过使用 `Context` 以外的方法获取到 `SType` 的示例，这也同时能够使类型的比较只需要比较进行指针比较而不需要通过复杂的递归。同时，建立了此类型系统与 LLVM 类型系统之间的映射。

AST2 的结构类似于 AST1， 但在 `Expression` 上标注了结果的类型，用以检查 AST2 结构的合理性。对于非法的整数与浮点数之间的类型转换会直接抛出异常。标注了类型的 AST2 在转换到 LLVM IR 会更加方便。具体的实现细节会在后文中提到。

编译器命令行命令解析

这里我们直接使用了 LLVM 的 `Support` 模块中的命令行工具，在全局作用域中声明用作命令行参数的 flag：

```
cl::opt<string> OutputFilename("o", cl::desc("Generate executable file with llvm and clang facilities"), cl::value_desc("output filename"));
cl::opt<bool> GenerateIR("emit-ir", cl::desc("To generate LLVM IR"));
cl::opt<bool> GenerateDot("emit-dot", cl::desc("To generate dot file"));
cl::opt<bool> GenerateCpp("emit-cpp", cl::desc("To generate cpp file"));
cl::opt<bool> GeneratePng("emit-png", cl::desc("To generate png file by dot"));
cl::opt<bool> PrintAst("emit-ast", cl::desc("Print abstract syntax tree in console"));
cl::opt<bool> Execute("i", cl::desc("Just-in-time execute with lli"));
cl::opt<string> InputFilename(cl::Positional, cl::desc("<input file>"), cl::Required);
```

然后通过其命令行解析器来获取每个选项的值或是输出帮助选项：

```
cl::ParseCommandLineOptions(argc, argv);
```

编译器前端

词法分析

SGS 词法分析由确定有限状态机完成。根据 SGS 语言特征，将输出的 token 流分为四类：关键字、用户标识符、操作符以及字面常量。

词法分析器读入 SGS 源代码，将输入流连接至词法分析器，然后输出语法分析器所需的 token 流。

确定有限状态机一共分为起始状态（接受状态）、读入标识符状态、读入数字状态以及读入符号状态。起始状态与接受状态等价，所以合并为一个状态。当读入一个字符时，进入读入标识符状态，当读入一个数字时，进入读入数字状态，当读入一个符号时，进入读入符号状态。读入标识符状态可能输出关键字或者用户标识符，读入数字状态可能输出数字常量，读入符号装固态可能输出符号或者字符串常量。

为了加速词法分析的过程，在字符串匹配的时候分析器使用了哈希算法进行插入与查找，提高了词法分析的效率。

语法分析

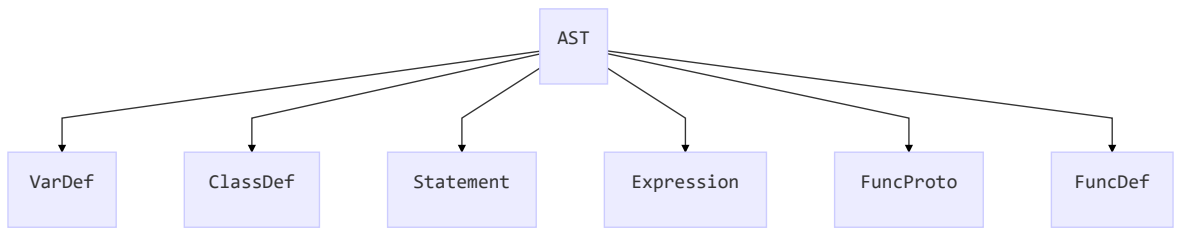
SGS语法分析使用LL(1)递归下降分析算法完成。

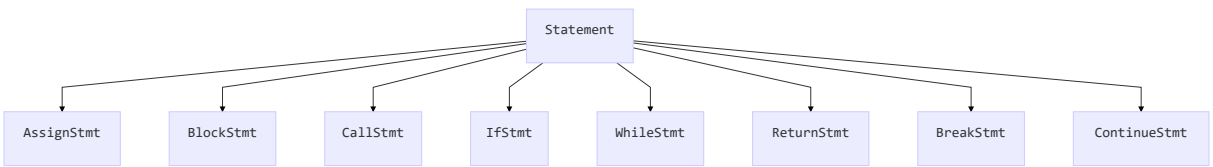
首先定义数据结构。

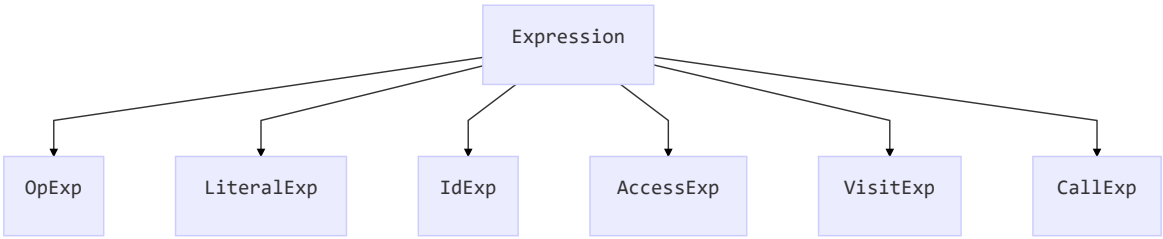
对于类型系统，我们定义 `class VarType` 为所有类型的基类。然后，定义三种继承类：`class BasicType` 、 `class ArrayType` 、 `class ClassType` 。其中BasicType为基础类型，包括整形、浮点型、布尔型、字符型和字符串型。ArrayType为数组类型，ClassType为结构类型。由于SGS中我们避免了指针的使用，所以在赋值及参数传递的时候，只有基础类型会按值传递，数组和结构均按引用传递。这样的好处在于代码更加安全，不存在直接访问随机地址的可能性。

对于AST，我们定义 `class AST` 为抽象语法树所有节点的基类。对于该抽象语法树基类，我们又如下继承类： `class VarDef` (变量声明)、 `class ClassDef` (类声明)、 `class Statement` (语句)、 `class Expression` (表达式)、 `class FuncProto` (函数声明)、 `class FuncDef` (函数定义)。其中Statement类包含更多继承类： `class AssignStmt` (赋值语句)、 `class BlockStmt` (语句块)、 `class CallStmt` (函数调用语句)、 `class IfStmt` (分支语句)、 `class WhileStmt` (循环语句)、 `class ReturnStmt` (返回语句)、 `class BreakStmt` (跳出语句)、 `class ContinueStmt` (重做语句)。Expression类也包含很多继承类： `class OpExp` (二元操作符表达式)、 `class LiteralExp` (字面量表达式)、 `class IdExp` (用户标识符表达式)、 `class Access` (类成员表达式)、 `class VisitExp` (数组元素表达式)、 `class CallExp` (函数调用表达式)。

如此定义SGS的AST数据结构，对其多态操作非常有利。在处理的过程中很多节点可以进行上溯造型，将其作为父类节点指针进行操作。这样，语法分析部分的代码逻辑更加清晰，更加容易理解。







对于顶层语法分析，可用如下伪代码表示：

```
switch(token){
    case new:
        declare();
        break;
    case start:
        function();
        break;
    case let:
        exp();eat(be);exp();
        break;
    case id:
        id();param();
        break;
    case if:
        exp();eat(then);block();eat(else);block();eat(end if);
        break;
    case loop:
        eat(when);exp();block();
        break;
}
```

因为顶层代码空间只支持类与函数的定义以及赋值、函数调用、分支和循环语句，所以为这几种情况分配case进行递归下降分析。在分析过程中，会递归用到declare（声明函数或类）、function（函数定义）、exp（表达式）、id（用户标识符）、param（函数参数）、block（语句块）等等。

其中会继续递归下降的操作为exp、block。block递归方式与顶层空间类似，不过其禁止函数和类的定义。即，SGS仅支持全局函数和类的声明。exp的语法分析可用如下伪代码表示：

```
stack values;
stack ops;
switch(token){
    case id:
        values.push(id);
        break;
    case data:
        values.push(data);
        break;
    case op:
        while(op > ops.top()){
            op1 = ops.pop();
            value1 = values.pop();
            value2 = values.pop();
            values.push(value1 op1 value2);
        }
        ops.push(op);
        break;
    case call:
        value.push(call());
        break;
}
```

这样，就完成了语法分析，将输入的token流转化为了AST1。

在错误恢复方面，SGS采用了书中的替换策略，即对于可恢复的错误，选择可用的语法来分析更正过的代码，然后增加一条warning消息。对于不可恢复的错误，跳过当前分析行，然后增加一条error消息。这样，代码中的语法错误就不会阻断编译的过程，编译器会扫描全部代码之后给出全部的错误，而非仅仅第一条错误。

可恢复的错误如漏掉某特定关键字，或者start和end之后的名称不匹配，这些错误可通过编译器直接替换来完成，这个时候给出warning告诉用户该处进行了某种替换即可。不可恢复的错误如连续多个用户标识符，SGS中无此语法，也无法进行自动替换，这个时候，分析器跳过当前行，给出错误信息。虽然，这样跳过当前行的做法可能会漏掉一小部分语义，但是对后续的语法分析并没有影响，于是SGS最终采取这种错误处理方法。

编译器后端

语义分析

经过编译器前端得到的 AST1 中仅仅保存了每个出现的标识符的名字，在语义分析阶段中，我们将对每一个词法作用域（在这里包括所有的 `Block` 和全局空间）建立一个符号表 `Env` 用来保存变量的类型：

```
class Env {
    Env* parent;
    std::map<string, sgs_backend::SType*> bindings;
public:
    Env(Env* parent) : parent(parent) {}
    void insert(const string& str, sgs_bakend::SType*);
    sgs_backend::SType* find(const string& str);
};
```

由于词法作用域本身是一个嵌套的结构，所以每个环境都会保存它的父环境用来实现跨域访问。同时，通过这种方法也能够实现不同作用域的变量 shadow 。

语义分析是与 AST 转换同时进行的，由于 SGS 本身支持全局作用域下的 `Statement` ，而 LLVM IR 则是通过 `main` 函数入口来执行代码，所以这里需要创建一个单独的 `main` 函数，并将顶层空间中的 `Statement` 全部放在 `main` 函数的 `body` 中。同时，为了消除 `result` 作为函数返回值的语法糖，在创建函数的时候，会在函数的 `body` 的头部插入一个变量定义，并对所有的 `Return Statement` 增加 `result` 这个值。

类型检查

语义分析中的一个关键就是对于类型的检查，由于运算符与函数调用被显式地分开了，所以这里需要完成的类型检查包括：

- 运算符类型检查
- 数组下标访问表达式类型检查
- 结构体成员访问表达式类型检查
- `IfStatement` 条件类型检查
- `WhileStatement` 条件类型检查
- 函数调用参数类型检查

因为 SGS 支持了整数类型之间的隐式类型转换，而没有支持浮点数和整数类型之间的隐式转换，所以在对运算符表达式进行类型检查时需要对浮点数和整数的运算抛出异常；`VisitExp` 需要保证访问的主体的类型为数组，而下标的类型必须为 `Integer` ； `AccessExp` 则要在结构体的复合类型中寻找访问的成员名字，如果没有找到也需要抛出异常。而函数的

调用中一个需要考量的点是数组参数的处理方法。在检查类型是否匹配时将只检查数组元素类型而不考虑数组长度。

针对类型转换，我们对二元运算符得到的结果类型单独使用了一个函数进行类型提升：

```
sgs_backend::SType* sgs_backend::getBinopType(BINOP op, SType* lhs, SType* rhs, Context& context) {
    switch (op) {
    case AND:
    case OR:
    case GT:
    case LT:
        return context.getBoolType();
    case ADD:
    case SUB:
    case MUL:
    case DIV: {
        assert(lhs->getLevel() == Types::BASIC_TYPE);
        assert(rhs->getLevel() == Types::BASIC_TYPE);
        const auto ls = dynamic_cast<SBasicType*>(lhs);
        const auto rs = dynamic_cast<SBasicType*>(rhs);
        if (ls->getBasicType() == BasicType::FLOAT) {
            return lhs;
        }
        if (ls->getBasicType() == BasicType::INTEGER || rs->getBasicType() == BasicType::INTEGER) {
            return context.getIntType();
        }
        if (ls->getBasicType() == BasicType::CHAR || rs->getBasicType() == BasicType::CHAR) {
            return context.getCharType();
        }
        return lhs;
    }
    default:
        return nullptr;
    }
}
```

代码生成

代码生成是编译器中最为核心的一部分，它揭示了从一门语言到另一门语言之间的映射关系。只有在对这两种语言都有着充分的了解之后才能找出这种映射关系。

我们选择的目标语言是 LLVM IR，所以这里我们会简单介绍 LLVM IR 的语法并讲解 LLVM C++ API 、LLVM IR 、AST2 之间的对应关系。

LLVM IR

> LLVM is a Static Single Assignment (SSA) based representation that provides type safety, low-level operations, flexibility, and the capability of representing 'all' high-level languages cleanly. It is the common code representation used throughout all phases of the LLVM compilation strategy.

一份 LLVM IR 代码由若干 `Module` 组成，对应着 C/C++ 中的一份（头）文件。每个 `Module` 中包含了一系列 Global Definitions， 包括各种函数声明/定义、全局变量定义、LLVM 元数据等。LLVM IR 和 C 语言类似，需要定义名为 `@main()` 的入口函数用以执行其内容。 LLVM IR 的 `Function` 由若干 `BasicBlock` 组成，每个 `BasicBlock` 会有一个入口 `label` ， 包含若干条 `Instruction` ，并且 `BasicBlock` 的最后一条 `Instruction` 一定是一条跳转指令，反之亦然。

LLVM IR 的类型系统作为一门底层的语言而说是非常强大的。 它支持任意 `bitsize` 的整数类型，数种小数类型，多个类型组成的 tuple ，以及数组、向量类型和指针等。足以覆盖我们的需求。在没有支持 垃圾收集 的情况下， LLVM IR 支持在栈上进行内存分配并得到相应类型的指针。

下面是一段典型的 LLVM IR 代码

```
declare i32 @sum(i32 %a, i32 %b) {
entry:
    %a.param = alloca i32
    %b.param = alloca i32
    store i32 %a, i32* %a.param
    store i32 %b, i32* %b.param
    %a.value = load i32, i32* %a.param
    %b.value = load i32, i32* %b.param
    %add.res = add i32 %a.value, %b.value
    ret i32 %add.res
}
```

与其对应的 C 语言代码是

```
int sum(int a, int b) {
    return a + b;
}
```

首先是 `declare i32 @sum(i32 %a, i32 %b)` 描述了函数的返回值类型，两个参数的类型和名字。

第二行的 `entry:` 声明了一个类型为 `label` ，名字为 `%entry` 的值作为函数的第一个 `BasicBlock` 的标签。

三四行使用 `alloca` 指令在 `@sum` 的栈空间中开出了两个 `i32` 的空间用来保存可变的形参 `a` 和 `b` 。

五六行使用 `store` 在刚才获取的空间中保存参数的值来为形参初始化。

下面开始对应于 C 语言中函数的正文： `return a + b;` 。这条语句中包含了两次从左值到右值的隐式转换，所以需要两次 `load` 指令来获取 `a` 和 `b` 的值。接下来通过 `add` 指令计算出它们的和，最终使用 `ret` 指令结束了这个 `BasicBlock` ，同时结束了这个 `Function` 。

可以看到，这里发生的变换是非常清晰、直接的，这对于我们生成从 AST2 生成 LLVM IR 也会带来很大的帮助。

使用 LLVM C++ API 生成代码

LLVM 本身提供了一套完善的 API 用来创建/优化/维护 LLVM IR 。这里我们不对其本身进行介绍，只会提到它的一些使用方法。

首先需要一个支持嵌套的符号表：

```
class Environment {
    Environment* parent;
    map<string, Value*> bindings;
    public:
    explicit Environment(Environment* parent = nullptr) : parent(parent) {}
    Value* operator [] (const string& str) {
        auto temp = this;
        while (temp) {
            if (temp->bindings.find(str) != temp->bindings.end()) {
                return temp->bindings[str];
            }
            temp = temp->parent;
        }
        return nullptr;
```

```
    }
    void insert(const string& str, Value* val) {
        bindings[str] = val;
    }
    static Environment* derive(Environment* env) {
        return new Environment(env);
    }
};
```

全局空间的 `Environment` 的 `parent` 是 `nullptr`，我们可以通过这个特征来判断变量定义的局部性。

然后是一些在代码生成过程中全程需要使用到的东西：

```
namespace sgs_backend {
using namespace llvm;
static map<string, Type*> typeReference;
static map<string, Function*> funcReference;
static LLVMContext theContext;
static Module* theModule;
static IRBuilder<> builder(theContext);
static Environment* globalEnv;
}
```

`LLVMContext` 在单个线程内用来管理其 `llvm::Type` 的资源获取和释放。因为 SGS 编译器暂时只支持单文件编译，所以这里只需要一个 `Module` 来存放所有的 Global Definition。`typeReference` 和 `funcReference` 用来保存顶层定义的类型和函数。 `builder` 是一个 `llvm::IRBuilder`，用来生成相应的指令。

对于顶层中会出现的语句，会分别按以下方法进行翻译：

1. 类型定义

顶层的类型定义仅包括结构体 (class) 定义，与其相对应的是 `llvm::StructType`，所以可以通过 `llvm::StructType::Create` 来在 `theContext` 中创建一个署名结构类型：

```
Type* toLLVMType(LLVMContext& context, const map<string, Type*>&
typeReference) const override {
    if (typeReference.find(name) != typeReference.end()) {
        return typeReference.find(name)->second;
    }
    vector<Type*> res;
    for (auto && type : types) {
        res.push_back(type.second->toLLVMType(context, typeReference));
    }
    return StructType::create(context, res, name);
}
```

2. 函数定义

函数的翻译需要先通过函数的原型 (Prototype) 得到函数的类型，在这里是一个 `llvm::FunctionType` ：

```
FunctionType* FuncProto::getLLVMType(LLVMContext& context, const map<string,
Type*>& typeReference) const {
    vector<Type*> res;
    for (const auto& x : paramList) {
        res.push_back(getParamType(x.first, context, typeReference));
    }
    return FunctionType::get(returnType->toLLVMType(context, typeReference),
res, false);
}
```

然后通过 `llvm::Function::Create` 来在 `theModule` 中创建一个新函数，并在 `funcReference` 中添加新函数

```
Function* fun = Function::Create(
    funType,
    GlobalValue::ExternalLinkage,
    funDef->getProto()->getName(),
    theModule
);
funcReference[funDef->getProto()->getName()] = fun;
```

紧接着为函数添加第一个 `BasicBlock` ，并命名其入口为 `entry` 、将 `builder` 的插入点设置为此 `BasicBlock`

```
BasicBlock* funBB = BasicBlock::Create(theContext, "entry", fun);
builder.SetInsertPoint(funBB);
```

由于结构体类型传参是传引用，在这里也就是 `const` 指针；而其他参数则需要单独声明一个变量，然后将作为函数参数的值存进来。

```
for (const auto& x : funDef->getProto()->getParam()) {
    if (x.first->getLevel() != Types::TUPLE_TYPE) {
        const auto temp = builder.CreateAlloca(iter->getType(), 0, nullptr, x.second);
        env->insert(x.second, temp);
        builder.CreateStore(iter, temp);
        iter++;
    } else {
        iter->setName(x.second);
        env->insert(x.second, iter);
        iter++;
    }
}
```

最后则开始翻译函数的主体，也就是一个 `BlockStatement`

3. 全局变量定义

为了使数组参数不会因为长度而被限制，生成的 LLVM IR 中所有与数组相关的操作都将会使用数组头指针代替整个数组，这样使得每个数组的定义都需要单独定义另外一个变量，用来保存数组头指针。

```
if (tp->getLevel() == Types::ARRAY_TYPE){ // complex type global variable definition
    const auto aryTp = dynamic_cast<SArrayType*>(tp);
    Constant* res = new GlobalVariable(
        *theModule,
        aryTp->toLLVMType(theContext, typeReference),
        false,
        GlobalValue::CommonLinkage,
        ConstantAggregateZero::get(aryTp->toLLVMType(theContext, typeReference)),
        glbVarDef->getName() + ".array");
    Constant* get = ConstantExpr::getInBoundsGetElementPtr(
        aryTp->toLLVMType(theContext, typeReference),
```

```
        res, vector<Constant*>({
            Constant::getIntegerValue(Type::getInt32Ty(theContext), APInt(32, 0)) ,
            Constant::getIntegerValue(Type::getInt32Ty(theContext), APInt(32, 0))
        }));
    const auto temp = new GlobalVariable(
        *theModule,
        aryTp->getElementType()->toLLVMType(theContext, typeReference)->getPointerTo(0),
        false,
        GlobalValue::InternalLinkage,
        get,
        glbVarDef->getName());
    return globalEnv->getBindings()[glbVarDef->getName()] = temp;
}
```

生成的所有全局变量都将用 0 进行初始化。

考虑到对于字符串字面量的支持，我们在翻译时开了一个洞。我们设置了一个单独的 AST 类型用来表示常量字符串，它仅能出现在赋值表达式的右边，且左边是一个长度不小于其的 `char` 数组。这一条语句会被翻译成一次对 `strcpy` 的调用：

```
if (ass->getRigth()->getExpType() == ET_CONSTR) { // deal with string assignment seperately
    Value* lhs = exprCodegen(ass->getLeft(), env);
    string conStr = dynamic_cast<ConstString*>(ass->getRigth())->getStr();
    Type* strType = ArrayType::get(Type::getInt8Ty(theContext), conStr.length() + 1);
    Constant* constStr = ConstantDataArray::getString(theContext, conStr);
    GlobalVariable* constStrv = new GlobalVariable(
        *theModule,
        strType,
        true,
        GlobalValue::PrivateLinkage,
        constStr,
        conStr + ".str"
    );
    Value* strPtr = builder.CreateLoad(lhs, "str.ptr");
    Value* conStrPtr = builder.CreateInBoundsGEP(
        constStrv,
```

```
        {
            Constant::getIntegerValue(Type::getInt32Ty(theContext), APInt(32, 0)) ,
            Constant::getIntegerValue(Type::getInt32Ty(theContext), APInt(32, 0))},
        "const.str.ptr"
    );
    return builder.CreateCall(funcReference["strcpy"], { strPtr, conStrPtr });
}
```

整数类型的隐式转换在二元运算符的翻译过程中随处可见，这里使用了一个函数用来将运算符两边的变量提升到同一个整数位宽上：

```
inline void integerTypeExtension(Value*& lhs, Value*& rhs) {
    const auto tl = dyn_cast<IntegerType>(lhs->getType());
    const auto tr = dyn_cast<IntegerType>(rhs->getType());
    if (tl->getBitWidth() < tr->getBitWidth()) {
        lhs = builder.CreateSExt(lhs, tr, "sext.temp");
    } else if (tl->getBitWidth() > tr->getBitWidth()) {
        rhs = builder.CreateSExt(rhs, tl, "sext.temp");
    }
}
```

对于控制流语句 `IfStatement` 和 `WhileStatement` ，只需要创建新的 `BasicBlock` 和 有/无条件跳转语句即可完成翻译。下面是用来翻译 `IfStatement` 的代码。首先通过 `builder` 来获取当前插入点所在的函数，然后创建 3 个 `IfStatement` 需要的 `BasicBlock` 。第一个 `BasicBlock` 可以直接接在当前函数的 `BasicBlock` 表后，而其它的两个需要在翻译完相应的分支内容后再插入。

```
case ST_IF: {
    Function* fun = builder.GetInsertBlock()->getParent();
    const auto ifs = dynamic_cast<IfStmt*>(stmt);
    Value* cond = exprCodegen(ifs->getCond(), env);
    if (cond->getType()->isPointerTy()) { // implicit cast from lvalue to rvalue
        cond = builder.CreateLoad(cond, "if.cond.load");
    }
    BasicBlock* taken = BasicBlock::Create(theContext, "if.take", fun);
    BasicBlock* untaken = BasicBlock::Create(theContext, "if.fail");
```

```
    BasicBlock* merge = BasicBlock::Create(theContext, "if.merge");
    builder.CreateCondBr(cond, taken, untaken);
    builder.SetInsertPoint(taken);
    stmtCodegen(ifs->getPass(), env, cont, bk);
    builder.CreateBr(merge);

    fun->getBasicBlockList().push_back(untaken);
    builder.SetInsertPoint(untaken);
    stmtCodegen(ifs->getFail(), env, cont, bk);
    const auto res = builder.CreateBr(merge);

    fun->getBasicBlockList().push_back(merge);
    builder.SetInsertPoint(merge);

    return res;
}
```

再者是局部变量的定义时的处理方法。

因为内存管理的关系，所有的局部变量声明（内存分配）都需要调整到函数的第一个 `BasicBlock` 中，防止无意义的重复内存分配。

```
case ST_VARDEF: {
    const auto vardef = dynamic_cast<VarDefStmt*>(stmt);
    IRBuilder<> tempBuilder(&builder.GetInsertBlock()->getParent()->getEntryBlock(), builder.GetInsertBlock()->getParent()->getEntryBlock().begin());
    Value* res = tempBuilder.CreateAlloca(vardef->getVarType()->toLLVMType(theContext, typeReference), nullptr, vardef->getName());
    Type* type = vardef->getVarType()->toLLVMType(theContext, typeReference);
    if (type->isArrayTy()) { // we shall use the array type with its element pointer
        const auto* atype = dyn_cast<ArrayType>(type);
        IRBuilder<> tempBuilder2(&builder.GetInsertBlock()->getParent()->getEntryBlock(), builder.GetInsertBlock()->getParent()->getEntryBlock().begin());
        Value* res2 = tempBuilder2.CreateAlloca(PointerType::get(atype->getElementType(), 0), nullptr, vardef->getName() + ".ptr");
        res = builder.CreateInBoundsGEP(res, {
            Constant::getIntegerValue(Type::getInt32Ty(theContext), APInt(32, 0)),
```

```
            Constant::getIntegerValue(Type::getInt32Ty(theContext),
APInt(32, 0))
        });
        builder.CreateStore(res, res2);
        res = res2;
    }
    env->insert(vardef->getName(), res);
    if (vardef->getInitValue()) {
        builder.CreateStore(exprCodegen(vardef->getInitValue(), env), res);
    }
    return res;
}
```

对于数组类型的变量声明，会通过一次显式的 `getelementptr` 指令获取数组的头指针，并将其作为一个值来分配一个新的变量以保存这个值，如果用 C 语言来描述的话大约是这样：

```
{
    int a[10];
    a[0] = 1;
    /* after transform */
    int _a[10];
    int* a = _a;
    a[0] = 1;
}
```

这个措施主要是使数组作为函数参数时能得到形式上的统一。

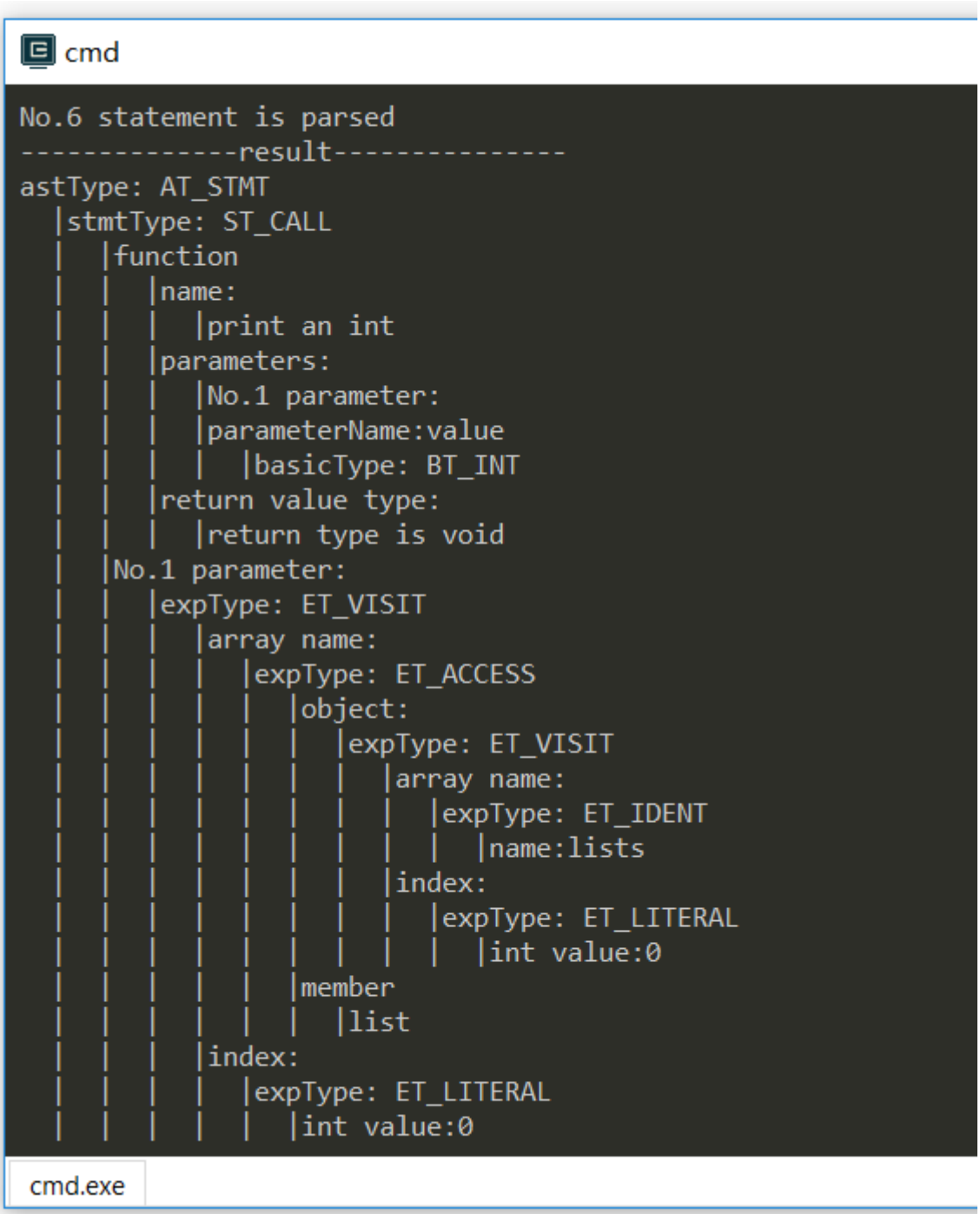
内置函数会在 `funcReference` 中保留函数类型定义，然后在生成代码时粘贴到代码开头。最终使用 `llvm::raw_fd_ostream` 将 `Module` 中的内容输出到指定的文件中。

辅助工具

命令行 AST 输出

单纯借助IDE（我们使用的是Visual Studio 2017）调试AST1将会困难重重，因为AST树已经存放在内存中，监视某一个AST树的结点变量需要进行多次强制变量转换才能获取所有细节。这种调试方法既繁琐，又十分容易在调试过程本身犯错。

因此为了让调试AST1的过程更为顺利，同时为了保证调试过程本身不出现错误，我们编写了一个命令行输出AST1树的调试工具 `printAST`。具体效果如下：



本句SGS语言源码为： `print an int with value lists[0]'s list[0].`

该工具遵循以下逻辑工作（工作思想类似于递归下降法）：

1. 将通过语法分析拿到的AST“ 森林” 传入工具主函数，该森林存储在`vector<sgs::AST *>`这一容器中；
2. 主函数依次访问容器中的每颗AST树，并分析每颗AST树；
3. 主函数在分析一颗AST树时，将先分析该树的根结点的类型`sgs::astType`，以判断当前AST树是什么类型，然后进入不同的处理分支作更进一步分析；

4. 在不同的子处理分支中，当前被分析的根结点将会被`dynamic_cast`为对应分支的类型，并调用相应的子处理函数打印根结点信息。在这之后子处理函数将继续分析根结点的2个child节点，过程类似于步骤3与步骤4。

上述过程将递归进行，直至分析到最底层的AST结点（例如`sgs::BasicType`的对象）为止。

举例说明：如上图，工作过程如下：

1. 本调试工具会首先分析当前AST树的根结点，发现AST种类是`sgs::AST_TYPE::AT_STMT`，调试工具就会进入相应的处理分支；
2. 调试工具调用`void dealWithStmtType(sgs::AST *s);`处理当前结点. 将当前结点转化为`sgs::Statement *`后，检查`sgs::Statement::stmtType`，发现当前结点具体种类为`sgs::STMT_TYPE::ST_CALL`；
3. 调试工具于是打印`stmtType: ST_CALL`，并设置缩进增加一格，然后分析当前结点的左child结点与右child结点，重复类似于步骤1、2、3的工作；
4. 当调试工具递归分析到最底层的AST结点后，将只打印最底层的AST结点的信息，而不是试图继续分析左右child结点（因为不存在）；
5. 当调试工具遵循上述4个步骤遍历完一整颗AST树后，该AST树的具体结构也通过命令行的形式可视化的打印出来，此时调试工具将转向分析下一棵AST树，直至`vector<sgs::AST *>`中所有的AST树均被遍历完毕。

借助上述调试工具，我们可以很清晰的看出存放在内存中的AST的结构，当AST存在错误的时候，可以一眼看出错误所在点，进而排除对应bug。

生成可执行的 C++ 代码

作为一个附加功能，我们还编写了一个附加工具：将SGS语言转化为C语言的工具`C-Codegen`。

该工具转化出来的代码虽然是C++，但是转化完成代码仅使用了C++的部分语法糖，实质上仍然是个C代码。

考虑到SGS语言在表面上更接近于python，转化成C代码时必须遵循下列规则转化：

1. 转化最开始，需要插入引用的头文件。为了方便起见，我们选择了一次性插入所有可能会用到的头文件；
2. 插入可能要用到的所有`using`；
3. 插入所有SGS内置的库函数对应的C语言翻译
4. 顶层的变量都是全局变量；
5. `int main()`函数外放置`class`的声明、`function`的声明和定义；
6. `int main()`函数内只能有表达式、语句。

如果要进行语言转化，必定需要了解所给定AST树的所有结点的信息，而在命令行输出工具`printAST`正好拥有遍历AST树所有结点的功能。因此我们编写的该工具是以`printAST`的代码框架为基础，进行一定程度的改动编写而成的。

与`printAST`不同，我们不需要打印AST树各结点信息，而是需要通过一定顺序访问AST树所有结点，并且每访问一个结点，我们就要将当前结点的信息以适当的方式插入到输出的`.cpp`文件中。

该转化工具以如下逻辑工作：

1. 由于需要遵循规则转化（见规则5与规则6），所以本工具需要循环遍历两次AST“森林”，第一次循环转化应该放在`int main()`函数外的语句，第二次循环转化放在`int main()`函数内的语句。工具根据每棵AST树的根结点的类型`sgs::astType`决定其应该在第一个循环还是第二个循环转化；

2. 对于每个AST树，遍历每个结点，获取信息，再以适当的C语法规定的顺序排列，输出到`.cpp`文件中。

转化样例：

```
new class list with integer name, integer array 10 list.
let list one.
let one's list[0] be 1.
print an int with value one's list[0].
let list array 10 lists.
let lists[0]'s list[0] be 2.
print an int with value lists[0]'s list[0].
```

生成的代码为:

```
#include <iostream>
#include <string>
#include <cstdio>
using std::string;

void print_an_int(int a){
    printf("%d\n", a);
}

void print_a_number(double a){
    printf("%lf\n", a);
}
```

```
13
14  void print_a_bool(bool a){
15      if(a == true)
16          std::cout << "true" << std::endl;
17      else
18          std::cout << "false" << std::endl;
19  }
20
21  void print_a_str(std::string a){
22      std::cout << a << std::endl;
23  }
24
25  float intToFloat(int a) {
26      return a;
27  }
28
29  int floatToInt(float a) {
30      return a;
31  }
32
33  class list{
34  public:
35      int name;
36      int list[10];
37  };
38  list one;
39  list lists[10];
40  int main() {
41      one.list[0] = 1;
42      print_an_int(one.list[0]);
43      lists[0].list[0] = 2;
44      print_an_int(lists[0].list[0]);
45      return 0;
46  }
```

生成 DOT 文件得到 AST2 图形显示

graphviz 是一款优秀的框图生成软件，其使用的描述图结构的语言为 DOT 。抽象语法树的结构非常清晰，非常适合与使用 graphviz 进行可视化。

由于生成的是有向图，所以在开头需要标注 `digraph` 。

对于 AST 在图上的表示，我们会把 AST 的属性的内容作为图上的一个节点，而属性的含义作为图上的边进行标注。为了保证生成的每一个节点的唯一性，我们会使用 AST 节点的地址作为在 图上的标注

下面是生成的一个例子：

sgs 代码为:

```
let integer a be 0.
print an int with a.
```

翻译到 C 后为：

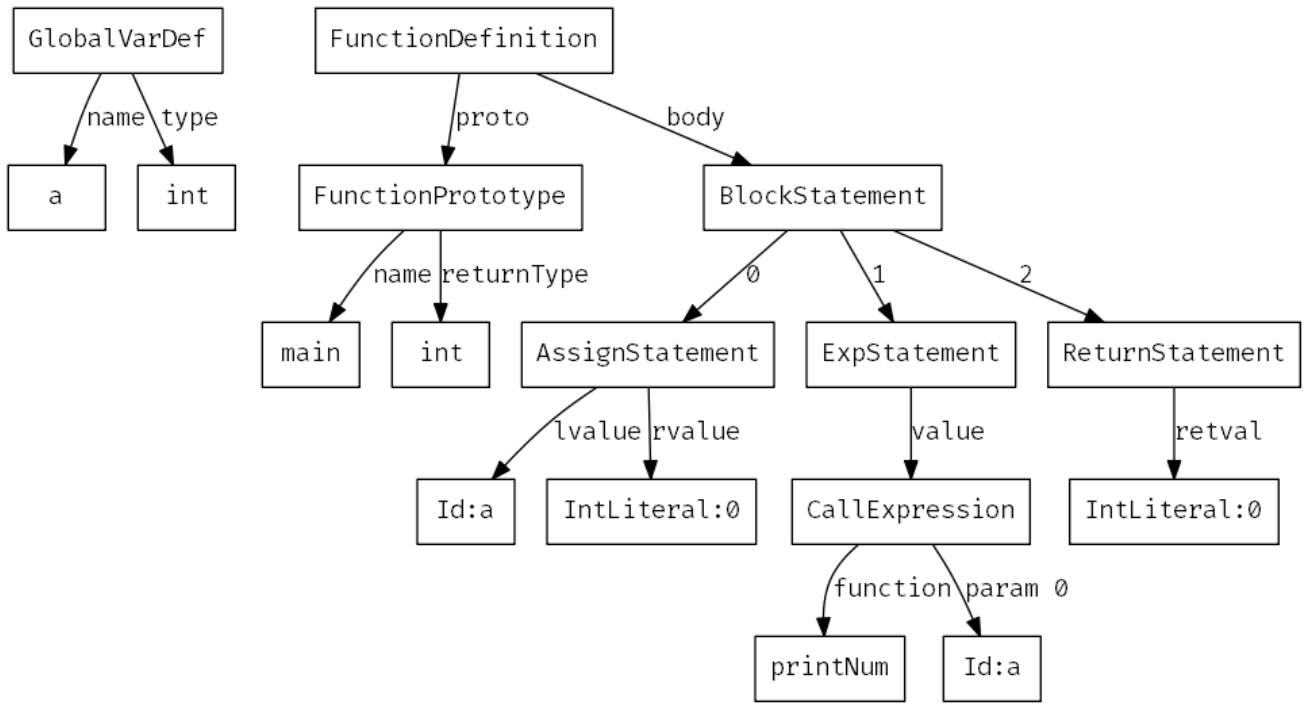
```c
int a;
int main() {
    a = 0;
    printNum(a);
    return 0;
}
```

生成的 DOT 文件为：

```
digraph g {
node[shape = box, fontname = "Fira Code Light"]
edge[fontname = "Fira Code Light", splines = line]
54691248 [label="GlobalVarDef"]
54691248 -> 1 [label="name"]
1 [label="a"]
54691248 -> 2 [label="type"]
2 [label="int"]
54713400 [label="FunctionDefinition"]
54713400 -> 54702592 [label="proto"]
54713400 -> 54667824 [label="body"]
54702592 [label="FunctionPrototype"]
54702592 -> 3 [label="name"]
3 [label="main"]
54702592 -> 4 [label="returnType"]
4 [label="int"]
54667824 [label="BlockStatement"]
54667824 -> 54712632 [label=0]
54667824 -> 54713528 [label=1]
```

```
54667824 -> 54712504 [label=2]
54712632 [label="AssignStatement"]
54712632 -> 54692216 [label="lvalue"]
54712632 -> 54712440 [label="rvalue"]
54692216 [label="Id:a"]
54712440 [label="IntLiteral:0"]
54713528 [label="ExpStatement"]
54713528 -> 54703112 [label="value"]
54703112 [label="CallExpression"]
54703112 -> 5 [label="function"]
5 [label="printNum"]
54703112 -> 54690808 [label="param 0"]
54690808 [label="Id:a"]
54712504 [label="ReturnStatement"]
54712504 -> 54713912 [label="retval"]
54713912 [label="IntLiteral:0"]
}
```

对应的图片是



测试

简单测试

整数/布尔类型变量声明和赋值，If 语句功能测试

SGS 代码

```
let integer a be 0.
let integer b be 1.
let bool c be false.
let b be b+a.
let c be b>a.
if c then let b be 5.
else let b be 3.
end if.
print an int with a.
```

浮点数整数转换、While 语句功能测试

```
let integer i be 1.
let float fact be 1.0.
let float res be 1.0.
loop when i < 11
    let res be res + 1.0 / fact.
    let fact be fact * intToFloat with value i.
    let i be i + 1.
end loop.
print a number with value res.
```

数组定义/访问测试

```
#localArrayTest
let integer array 10 a.
let a[0] be 0.
let a[1] be a[0] + 5.
print a number with value a[1].
```

全局变量/数组访问测试

```
let integer a.
let integer array 10 arr.
new function visitGArray with integer i return integer.
start visitGArray.
    let result be arr[i].
end visitGArray.
new function writeGArray with integer array 10 b, integer i, integer v
return integer.
start writeGArray.
    let b[i] be v.
end writeGArray.
writeGArray with arr, 1, 2.
print a number with visitGArray with 1.
```

字符串字面量赋值/修改测试

```
let char array 4 x.
let x be "123".
let x[0] be x[0] + 1.
print a str with value x.
```

结构体内部数组/结构体数组测试

```
#complex type
new class list with integer name, integer array 10 list.
let list one.
let one's list[0] be 1.
print an int with value one's list[0].
let list array 10 lists.
let lists[0]'s list[0] be 2.
print an int with value lists[0]'s list[0].
```

复杂测试

这里使用 SGS 实现了一个递归下降的简单数学表达式计算器，仅保证针对正确的输入会得到正确的输出

对于输入中的所有空格会直接忽略，当遇到换行时输出结果

```

```

```
let char array 100 str.
let integer length.
let integer position.

new function current return char.
start current.
    if position <= length then
        let result be str[position].
        return.
    end if.
    let result be 0.
end current.

new function match.
start match.
    let position be position + 1.
end match.

new function parseNum return integer.
start parseNum.
    let char c be current.
    let result be 0.
    loop when c >= 48 && c <= 57.
        let result be result * 10 + c - 48.
        match.
        let c be current.
    end loop.
end parseNum.

new function parseFactor return integer.

new function parseTerm return integer.
start parseTerm.
    let result be parseFactor.
    let char c be current.
    loop when c = 42 || c = 47.
        if c = 42 then
            match.
            let result be result * parseFactor.
        else
            match.
            let result be result / parseFactor.
```

```
        end if.
        let c be current.
    end loop.
end parseTerm.

new function parseExpr return integer.
start parseExpr.
    let result be parseTerm.
    let char c be current.
    loop when c = 43 || c = 45.
        if c = 43 then
            match.
            let result be result + parseTerm.
        else
            match.
            let result be result - parseTerm.
        end if.
        let c be current.
    end loop.
end parseExpr.

start parseFactor.
    let char c be current.
    if c = 40 then
        match.
        let integer temp be parseExpr.
        match.
        let result be temp.
        return.
    end if.
    let result be parseNum.
end parseFactor.

loop when true.
    let length be 0.
    let position be 0.
    let char c be getchar.
    loop when c != 10.
        if c = 32 then
            let c be getchar.
            redo.
        end if.
```

```
        let str[length] be c.
        let length be length + 1.
        let c be getchar.
    end loop.
    let integer res be parseExpr.
    print an int with value res.
    newline.
end loop.
```

后记

分工

编译器开发组成员及工作

- 朱瑞昇：SGS 语法设计、编译器前端部分及其对应文档、测试样例的完成
- 蔡展璋：编译器前端测试以及调试，辅助调试工具、CPP 代码生成器及其对应文档
- 魏耀东：编译器后端、DOT生成器部分及其对应文档，对编译器整体的测试

扩展与改进

- [ ] 支持 `import` 和 `export` 等实现代码的模块化
- [ ] 支持类成员方法，引入 `this` 关键字
- [ ] 实现 SGS 标准库
- [ ] 实现与 C 语言之间的互操作，引入和使用 C 语言的文件/方法。
- [ ] 实现简单的宏