Отчет

Расширенный проект сборки

Разработка интерпретатора языка UM универсальной машины

Разработка компилятора с языка S-UM на язык UM

Амель АРКОУБ 3301571

Линг-Чунь СО 3414546

04 апреля 2018 г.

Резюме

0

Вступление

0.1 Презентация

Проект разделен на две части. Во-первых, мы предоставляем реализацию универсальной машины (www.boundvariable.org/um-spec.txt) из конкурса программирования Международной конференции ACM по функциональному программированию (ICFP) 2006 года (www.boundvariable.org/task.shtml). Мы также хотели бы иметь возможность программировать на языке, понятном универсальной машине, что непросто, поскольку это двоичный язык. Итак, на втором этапе мы напишем компилятор из спецификации универсального машинного языка S-UM в двоичный язык единой системы обмена сообщениями.

0.2 Описание файлов

Вот разные папки и что они содержат:

- ./um/ Эта папка содержит реализацию универсальной машины.
- ./sum/ Эта папка содержит реализацию компилятора S-UM.
- ./tests/ Эта папка содержит различные тесты компилятора S-UM.
- ./rapport.pdf Этот файл является самим отчетом.
- ./README.md Этот файл содержит инструкции по компиляции и тестированию проекта.
- ./rapport/ Эта папка содержит исходники этого отчета в LateX.
- ./archives/ Эта папка в основном содержит файлы, которые не следует рассматривать. Это начало анализатора/парсера на C для языка S-UM и реализации универсальной машины, манипулирующей списками массивов. Действительно, по соображениям производительности мы решили отказаться от него в пользу реализации, обрабатывающей массивы массивов.

0.3 Статус проекта

Проект в целом завершен, однако некоторые моменты требуют уточнения:

- универсальный станок (УМ): Функциональный
- компилятор S-UM: Функциональный
  Поддерживаются все функции языка, но важно указать:

– последовательность инструкций: Функциональная

– присвоение переменной: функциональное

- печать: можно улучшить

Действительно, печать работает как для строк, так и для констант, однако она отображает только вычисление по модулю 256. Например, печать 96+1 будет отображаться в выводе «a», где 97 — это буква «a» в ASCII.

- сканирование: улучшено

Сканирование занимает только один символ ASCII.

– альтернатива: Функциональная

– целое число: функциональное

– строка символов: функциональная

– арифметические выражения: функциональные – реляционные выражения: функциональные

– бинарные логические выражения: функциональные – унарные логические выражения: функциональные

1

Универсальная машина

1.1 Структура

На уровне реализации универсальной машины, поскольку каждая пластина кодируется 32 битами, мы используем 32-битное целое без знака:

```
typedef uint32_t uint32;
```

чтобы содержать инструкции программы, мы выбрали следующую структуру:

```
typedef struct array{размер
    uint32; uint32 *блюдо; }
    множество;
```

Это структура, используемая для хранения массива инструкций и размера памяти массива (последнее необходимо время выполнения инструкции ЗАГРУЗИТЬ ПРОГРАММУ).

Чтобы иметь возможность запомнить все многократно используемые индексы массива, у нас есть структура списка для индексов. Если список пуст, мы возвращаем глобальную переменную, которую увеличиваем.

```
внешний uint32 indexcpt; typedef
struct freeindex{
  индекс uint32;
  struct freeindex *следующий; }
свободный индекс;
```

1.2 Функции

Вот описание функций универсальной машины:

- array* loadFile(const char* filename) Возвращает массив, содержащий все инструкции, считанные из файла. имя файла в двоичном формате, затем инвертируйте порядок следования байтов (запуск виртуальной машины с файлом resources/sandmarkz.umz указывает, является ли порядок следования байтов неверным, отображая порядок следования байтов на консоли).
- freeindex* initFreeIndex() Инициализирует используемую очередь индексов.
- void addFreeIndex(freeindex** fi, uint32 index) Добавляет в очередь индексов fi индекс показатель.

- uint32 getFreeIndex(freeindex** fi) Извлекает пригодный для использования индекс. Если очередь пуста, мы возвращаем indexcpt++.
- void freeFreeIndex(freeindex** fi) Освобождает файловую структуру.
- array* initArray(uint32 size) Выделяет структуру массива размера size.
- void freeArray(array *arr) Удаление массива структур.

1.3 Интерпретация языка

Структура языковой интерпретации выглядит следующим образом:

```
регистры uint32[8] = {0}; в то
время как (1) { слово = ноль [pt]; оп
    = слово>>28; a = ((слово >> 6)
    и 0x7); б = ((слово >> 3) и 0x7);
    с = (слово & 0x7);
    переключатель (оп) {


        кейс ..:
        кейс ..:
        ...
    }
    пт++;
}
```

Мы инициализируем массив из 8 целых чисел без знака, соответствующих регистрам. Сердцем интерпретации является переключатель-кейс, содержащийся в бесконечном цикле. Перед переключением индексы операций и регистров извлекаются с помощью битовых сдвигов 32-битной инструкции. Наконец, после switch-case мы увеличиваем значение счетчика pt.

1.4 Спектакли

Не следует пренебрегать работой переводчика. Действительно, из-за нашей реализации на C с помощью таблицы, скомпилированной в -O3, мы имеем очень удовлетворительную производительность. Взяв файл sandmarkz.umz, мы получаем время 18,424 секунды, тогда как при реализации на JAVA выполнение программы может занять несколько минут.

1.5 Схема универсальной машины с списком

ПРИМЕЧАНИЯ: Мы собираемся обсудить первую реализацию универсальной машины, манипулирующей списками. Хотя он функционален, он очень медленный, и мы собираемся изучить его производительность. Поэтому следует отметить, что этот макет следует сохранять не для использования, а скорее для анализа. Однако, если вы хотите увидеть код, он находится в папке ./archives/Universal_Machine_list.

Первая версия универсальной машины основана на структуре списка. Хотя

эта реализация работает, но ее недостаток в том, что она очень медленная. Файл sand markz.umz по-прежнему не завершается после 10 часов выполнения. Итак, чтобы определить узкое место программы, мы использовали программу профилирования кода Gprof.
Этот инструмент используется для получения статистики о времени и количестве вызовов функций при выполнении программы. Чтобы иметь выходной файл с именем gmon.out, вы должны сначала отключить оптимизацию времени компиляции и скомпилировать с флагом -pg. Однако для создания этого файла программа должна завершиться корректно. Поэтому мы решили связать сигнал SIGUSR1 следующим образом:

```
#include <signal.h>
 ...
недействительным sig_exit
  () {выход(0); }

 ...
int main(int argc, char **argv){ signal(SIGUSR1,
  sig_exit);
  ...
}
```

После того, как файл gmon.out был сгенерирован (мы отправили сигнал SIGUSR1 в конце песочницы 100, применив команду:

```
gprof universal_machine gmon.out > analyse.txt
```

Получаем в файле analyse.txt:

Плоский профиль:

Каждый образец считается за 0,01 секунды.

% совокупный	собственный			себя	всего	
время	секунды	секунды		звонков	нам/позвонить	нам/позвонить по имени
99,52	146,74	146,74	5307119	147,60	0,86	27,65 getArray
0,58						main
0,03	147,65	0,05	44683	1.01	1.01	removeArray 0.12
0,01	147,66	0,01	87030	0,12		addArray 0.00
0,00	147,66	0,00	87030	0,00		getFreeIndex 0.00
0,00	147,66	0,00	44683	0,00		addFreeIndex
0,00	147,66	0,00	1	0,00		0.00 initArrays 0.00
0,00	147,66	0,00	1	0,00		initFreeIndex
0,00	147,66	0,00	1	0,00		0.00 файл загрузки

Таким образом, из 147,66 секунд выполнения почти 99,52% времени вычислений используется для выполнения функции getArray. Мы также достигаем очень большого количества звонков: 5307119 в течение относительно короткого времени. Таким образом, проблемы с производительностью были связаны с доступом к массиву, и именно тогда мы решили использовать реализацию, которая обрабатывала массивы массивов, а не реализацию, которая обрабатывала списки массивов.

2

Компилятор S-UM

2.1 Выбор инструментов

В этом проекте мы изначально решили написать компилятор на C с использованием инструментов Yacc и Flex. Однако в конечном итоге мы решили написать его на JAVA с использованием ANTLR 4.4. Хотя его написание занимает больше времени, его преимущество заключается в простоте использования. Старт старого компилятора C находится в папке ./archives/compiler_old_c. Компилятор, который мы используем и который будет описан позже, — это компилятор JAVA, использующий ANTLR, и он находится в папке ./sum.

2.2 Грамматика

Грамматика находится по пути ./sum/SUM/ANTLRGrammar.g4 и соответствует форме Бэкуса-Наура (BNF). Эта грамматика поможет обнаружить язык S-UM.

грамматика SUMграмматика;

prog возвращает [sum.interfaces.iast.IASTprogram node]:
(stmts+=stmt ';'?) * EOF
;

stmt возвращает [sum.interfaces.iast.IASTstatement node] : expr
#Expression | 'пусть' var=IDENT '='? значение=выражение?
#Привязка | 'печать' val=expr #Печать | 'scan' var=IDENT #Scan |
'if' cond=expr 'then' '{' (cons+=stmt ';'?)* '}' 'else' '{' (alt+=stmt ';'?)* '}'
#Alternative

;

expr возвращает [sum.interfaces.iast.IASTexpression node] :
intConst=INT # ConstInteger | stringConst=STRING # ConstString |
ident=IDENT #Идентификатор | arg1=expr op=('*' | '/' | '+') arg2=expr
#BinOp | arg1=expr op=('<' | '=' | '>') arg2=expr #RelationBinOp |
arg1=expr op=('И' | 'ИЛИ') arg2=expr #LogicBinOp | 'НЕ' arg=expr
#LogicUnOp;

```
INT: [0-9]+;
ИДЕНТИФИКАТОР: [a-zA-Z_] [a-zA-Z0-9_]*;
НИТЬ : '"' (ESC | ~["\\])* '"';
ESC: '\\' [\\nrt"];
LINE_COMMENT : '//' (~[\r\n])* -> пропустить;
КОММЕНТАРИЙ: '/*' ('*' ~[/] | ~[*])* '*/' -> пропустить;
ПРОБЕЛ: [ \t\r\n]+ -> пропустить;
```

Программа представляет собой набор операторов. Эти заявления либо:

- назначение
- печать
- сканирование
- альтернатива
- выражение

Выражения могут быть:

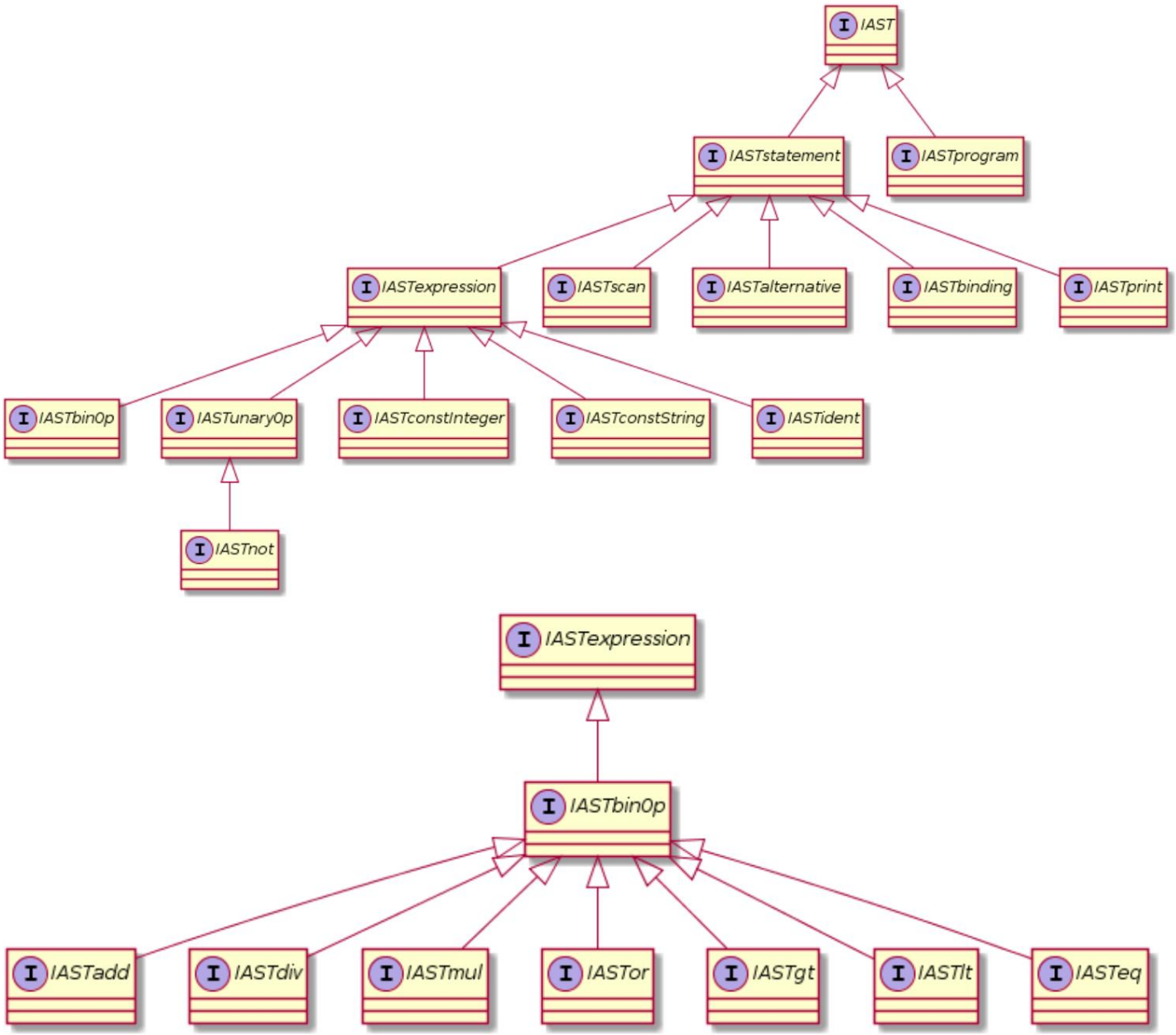
- целочисленная константа или строка символов
- идентификатор
- бинарная арифметическая операция между двумя выражениями
- операция бинарного отношения между двумя выражениями
- бинарная логическая операция между двумя выражениями
- унарная логическая операция между двумя выражениями

Выполнение инструмента ANTLR для этого файла создает несколько классов JAVA, позволяющих распознавать язык.

2.3 Абстрактное синтаксическое дерево

Чтобы сгенерировать абстрактное синтаксическое дерево, мы определили интерфейсы, а их реализации представляют собой простые контейнеры. Объявление интерфейса находится в папке ./sum/SUM/src/sum/interface, а классы реализации — в папке ./sum/SUM/src/sum/ast.

Рис унок 2.1: Диаг рамма клас с ов общей с труктуры



Клас с SUMListener реализует интерфейс SUMgrammarListener, он позволяет применить дейс твие к каждой рас познанной функц ии языка. Клас с SUMParser иниц иализирует анализатор ANTLR и выполняет правильную пос ледовательнос ть вызовов для пос троения абс трактног о с интакс ичес ког о дерева. Эти клас с ы дос тупны в папке ./sum/SUM/src/sum/parser.

2.4 Компиляц ия в язык S-UM

2.4.1 Пос етитель шаблона проектирования

Чтобы иметь возможнос ть прос матривать и обрабатывать каждый узел абс трактног о с интакс ичес ког о дерева, мы ис пользовали шаблон проектирования «Пос етитель», потому что он лучше вс ег о подх одит, ког да мы х отим выполнить интерпретац ию /компиляц ию .

на объектном языке. Таким образом, все узлы AST имеют общедоступный метод void accept(IASTvisitor visitor). Реализация класса Compiler используется для компиляции AST в язык.

2.4.2 Общее описание компиляции (компилятор классов)

Вот список переменных и описание их назначения:

- NO_CONTEXT Указывает, должна ли оценка, возвращенная во время визита, быть хранится или нет.
- numInst представляет количество записанных инструкций (что полезно для альтернативы).
- env Карта, позволяющая хранить в памяти имя переменной и индекс таблицы, в которой она должна храниться
- fos выходной поток

2.4.3 Хранение расчетов

Обратите внимание, что доступно всего 8 регистров, поэтому хранение промежуточных вычислений ограничено: нам нужно было найти способ хранения. Интересным свойством является то, что при загрузке программы занят только массив [0], поэтому мы знаем, что индекс свободных массивов начинается с 1. Затем, используя целочисленную переменную indVar, мы узнаем индекс массива, в котором будет выполнено следующее выделение массива, и в конце каждого выделения indVar увеличивается на 1. Важно отметить, что освобождение будет выполняться только при выходе из программы: при выходе из блока освобождение не происходит.
Для освобождения памяти при выходе блока индексы освобожденных массивов должны храниться в списке для последующего повторного использования во время выделения. Увы, мы не успели настроить эту реализацию. Каждый метод посещения имеет аргумент контекста int.
Если последний отличается от NO_CONTEXT, это индекс массива, в котором должен храниться результат.
Метод allocateVar() переводит инструкцию выделения размера 1 на язык единой системы обмена сообщениями, поскольку сохраняется только одно значение.

2.4.4 Преобразование в инструкцию UM

Непосредственный перевод в двоичный формат единой системы обмена сообщениями непрост, поэтому мы определили методы, облегчающие перевод. Вот описание:

- public void writeOperation(int op, int regA, int regB, int regC) Позволяет записать в выходной файл операцию op с регистрами a, b и c в двоичном формате UM, байты записываются с сдвигами. (Подробнее в файле компилятора в комментариях).
- public void writeSpecialOperation(int regA, int value) Позволяет загрузить беззнаковое 25-битное значение в регистр A. (Подробнее в файле компилятора в комментариях)
- public void fetchIntoReg(int i, int reg) Извлекает простое значение, находящееся в tree[i][0] (это значение переменной), и добавляет его в регистр reg.
- public void putIntoArray(int i, int reg) Добавляет в массив[i][0] значение в регистре с индексом reg.
- public void allocateVar() Выделяет плато размера 1 для переменной.

2.4.5 Равенство

Написание бинарных отношений представляло определенную трудность. Действительно, мы очень ограничены в возможных операциях . Однако нам это удалось. Во-первых , обратите внимание на следующее: пусть a будет целым числом a. NAND(a,a) инвертирует биты. 0 становятся1, а1 становятся0.

Таким образом, мы наблюдаем, что a + NAND(a,a) = 0b11..11, все биты установлены в 1.

Таким образом, добавляя этот результат к 1, мы превышаем 32 бита и, следовательно, возвращемся к 0b00..00.

Итак, a + НЕ -И(a, a) + 1 = 0b00..00.

Таким образом, мы можем проверить равенство a = b

следующим образом: если a + NAND(b,b) + 1 = 0b00..00, то a = b, иначе a != b.

Схема компиляции: context = (expr1 = expr2);

```
int contextarg1 = indVar++
allocateVar() int contextarg2 =
indVar++ allocateVar()
tableau[contextarg1][0] <- expr1
tableau[contextarg2][0] <- expr2 reg[0] <-
tableau[contextarg1][0] reg[1 ] <-
tableau[contextarg2][0] reg[2] <- NAND(1,1)
reg[3] <- 1 reg[2] <- reg[2] + reg[3] reg[0] <-
reg[0] + reg[2] reg[1] <- 1 reg[2] <- 0 reg[1]
<- если reg[0] = 0, то reg[1] иначе reg[2]
tableau[context] [0] <- регистр [1]
```

2.4.6 Отношение порядка: > и <

Для вычисления отношения порядка a > b достаточно произвести целочисленное деление. Если a/b > 0, то связь верна, но необходимо предпринять некоторые шаги. Мы также должны проверить случай, когда a = b, мы повторно используем код IASTeq. Более того, в случае, когда знаменатель b = 0, мы делим на 0. Чтобы избежать этого, рассматриваемый расчет будет (a+1)/(b+1). Это не меняет отношения, потому что если a/b > 0, то (a+1)/(b+1) тоже.

Схема компиляции: context=(expr1/expr2)

```
int contextarg1 = indVar++
allocateVar() int contextarg2 =
indVar++ allocateVar()


tableau[contextarg1][0] <- expr1
tableau[contextarg2][0] <- expr2 reg[1] <-
tableau[contextarg1][0] reg[2] <-
tableau[contextarg2][0]

// мы добавляем 1, чтобы избежать деления на 0
```

reg[3] <- 1
reg[1] <- reg[1] + reg[3] reg[2]
<- reg[2] + reg[3]

//если reg[1]/reg[2] = 0, то reg[1] < reg[2], потому что это целочисленное деление reg[3] <-
reg[1]/reg[2] reg[0] < - 0 reg[4] <- 1 reg[0] <- если reg[3] != 0 то reg[4] иначе reg[0]

//проверяем, что reg[1] != reg[2] //это те
же операции, что и для IASTeq reg[3] <- NAND(reg[2], reg[2])
reg[4] <- 1 reg[3] <- reg[3] + reg[4] reg[1] <- reg[3] + reg[1]
reg[2] <- 1 reg[3] <- 0

reg[2] <- если reg[1] != 0, то reg[3] иначе reg[2] reg[1] <- 0
reg[0] <- если reg[2] != 0, то reg[1] иначе регистр[0]

таблица[контекст][0] <- reg[0]

Код для операции < идентичен, за исключением того, что мы изменили:

tableau[contextarg2][0] <- expr1
tableau[contextarg1][0] <- expr2

2.4.7 Или

Перевод ИЛИ выполняется путем вычисления левого и правого членов. Используя операцию 0
(условное движение), мы определяем, равен ли результат 1 (истина) или ложь (0).

Схема компиляции: контекст = (выражение1 ИЛИ выражение2)

int contextarg1 = indVar++
allocateVar() int contextarg2 =
indVar++ allocateVar()
array[contextarg1][0] <- expr1
array[contextarg2][0] <- expr2 reg[1] <-
array[contextarg1][0] reg[2] <-
array[contextarg2][0] reg[0] <- 0 reg[3] <-
1 //reg[1] и reg[2] содержат левый и
правый элементы результата ИЛИ //
reg[0] содержит 0, если reg[1] или reg[2]
содержат значение != 0, // мы помещаем в reg[0] значение 1, так что это действительно ИЛИ
reg[0] <- if reg[1] != 0, затем reg[3], иначе reg[0]

```
reg[0] <- if reg[1] != 0 then reg[2] else reg[0] tableau[context]
[0] <- reg[0]
```

2.4.8 И

Перевод И очень похож на перевод Ор. Действительно, есть только модификация на уровне условных движений. Мы делаем «цепочку» условных движений, таким образом, reg[1] и reg[2] (регистрирует, что содержать результат вычисления выражений) должны оба быть ненулевыми, чтобы результат был равен 1 (истина).

Схема компиляции: контекст = (expr1 AND expr2)

```
int contextarg1 = indVar++
allocateVar() int contextarg2 =
indVar++ allocateVar()
array[contextarg1][0] <- expr1
array[contextarg2][0] <- expr2 reg[2] <-
expr1 reg[3] <- expr2 reg[ 4] <- 1 reg[5]
<- 0 reg[1] <- 0 reg[0] <- 0 //reg[2] и reg[3]
содержат результат левой и правой
частей AND // reg[1] = 0, если reg[2] !=
0, тогда reg[1] = 1 //reg[0] = 0, если
reg[3] != 0, тогда reg[0] = reg[1] //так
если у нас есть 0 И 1, reg[0] = 0 и т.д.
reg[1] <- если reg[2] != 0 тогда reg[4] else reg[1] reg[0] <- if reg[3] ! = 0 затем reg[1] иначе reg[0]
array[context][0] <- reg[0]
```

2.4.9 Не

Перевод Not в UM прост, выполняем условное перемещение: если значение отличается от 0 то 1, 0 иначе.

Схема компиляции: контекст = Not(expr)

```
int argcontext = indVar++
allocateVar() array[argcontext]
[0] <- expr reg[2] <- array[argcontext]
[0] reg[0] <- 1 reg[1] <- 0 //если
результат != 0 тогда мы ставим 1 иначе
0 reg[0] <- если reg[2] != 0 тогда reg[1]
else reg[0] tableau[context][0] <- reg[0]
```

2.4.10 Арифметическая операция

Арифметические операции соответствуют загрузке значения, применению операнда и сохранению результата.

Схема компиляции: context = (expr1 [+*/] expr2)

```
int contextarg1 = indVar++;
выделитьПеременную(); int
contextarg2 = indVar++;
выделитьПеременную();
tableau[contextarg1][0] <- expr1
tableau[contextarg2][0] <- expr2 reg[0] <-
tableau[contextarg1][0] reg[1] <-
tableau[contextarg2][0] reg[2] <- reg[0] [+*/]
reg[1] таблица[контекст][0] <- reg[2]
```

2.4.11 Альтернатива

Альтернатива имеет очень интересную компиляцию для изучения. Сначала мы попытались динамически вычислить количество пропускаемых инструкций для условий, но это было безрезультатно. Поэтому мы решили выделить блоки консеквента и альтернативы фиксированного размера, что упрощает определение адресных переходов. Короче говоря, альтернатива использует операции условного перемещения, чтобы определить, какой переход выполнить, а также операции загрузки программы, которые позволяют производить переходы команд. Мы используем здесь переменную numInst для определения адресов переходов, numInst увеличивается с каждой операцией записи.

Схема компиляции: если expr1, то expr2, иначе expr3

```
интервал DEFAULT_SIZE = 4096
int condctx = indVar++
allocateVar()
tableau[condctx][0] <- expr1

//в зависимости от значения условия переход или на адрес then //, либо на адрес else //блоки
then и else фиксированы, что позволяет вычислять //напрямую адреса переходов reg[1] <- numInst+8 //
затем reg[0] <- numInst+8+DEFAULT_SIZE-1 //иначе reg[2] <- array[condctx][0]




reg[0] <- если reg[2] != 0 тогда reg[1] иначе reg[0] reg[1] <- 0


//загружаем программу tableau[reg[1]] и указываем значение reg[0] //инструкция, которая будет
выполнятьсяна следующем ходу, будет tableau[reg[1]][reg[0]]
LOAD_PROG(рег[0], регистр[1])

//инструкции du then
```

```
int начало = numInst;
NO_CONTEXT <- expr2

//выйти из if reg[1] <-
0 reg[0] <-
start+2*DEFAULT_SIZE
LOAD_PROG(reg[0],reg[1])

//мы заполняем оставшуюся часть блока NOP'ами
while(numInst < start+DEFAULT_SIZE) { reg[0] <- 0

}

//инструкции вам еще
NO_CONTEXT <- expr3

//мы заполняем оставшуюся часть блока NOP'ами
while(numInst < start+2*DEFAULT_SIZE) { reg[0] <- 0

}
```

2.4.12 Привязка

Связывание представляет собой комбинацию объявления и присвоения переменной. Таким образом, мы добавляем переменную в среду компиляции, то есть в карту, имя и индекс массива, в котором она была назначена. Если переменная объявлена с выражением, вычисление выражения будет храниться там.

Схема компиляции: пусть var = expr

```
int varcontext = indVar++
allocateVar() env.put(var,
varcontext) if existsExpr() then
tableau[varcontext][0] <- expr
```

2.4.13 Идентификатор

Поскольку идентификаторы соответствуют именам переменных, мы храним в среде компиляции карту, содержащую имя переменной и индекс массива, в котором хранится переменная.

Схема компиляции: context = expr

```
int varcontext = env.get(expr) reg[0] <-
tableau[varcontext][0] tableau[context] <-
reg[0]
```

2.4.14 Целое число

Трансляция константы соответствует загрузке непосредственного значения в ячейку массива.

Схема компиляции: context = expr

```
reg[0] <- expr
tableau[context][0] <- reg[0]
```

2.5 Тесты

Наборы тестов находятся в папке ./tests, это модульные тесты каждой возможности языка. В основном мы проводим тесты с ожидаемыми дисплеями. Так как выражение отображается в форме ASCII, мы в большинстве случаев проверяем, соответствует ли буква, возвращаемая вызовом, ожидаемой букве. Доступны и другие более сложные тесты, но они не в запрошенном формате. Эти тесты можно найти в исходниках компилятора S-UM по пути ./sum/SUM/test.