Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Владимирский государственный университет
имени Александра Григорьевича и Николая Григорьевича Столетовых»
(ВлГУ)

РАЗРАБОТКА КОМПИЛЯТОРА

Пояснительная записка
На 16 листах

Руководитель _____ к.т.н. доцент кафедры ИЗИ Ю.М. Монахов

Исполнитель _____ студент гр. ИСБ-118 А.С. Соболев

Владимир 2021

Оглавление

Аннотация

В данной пояснительной записке приведено описание и примеры работы компилятора подмножества процедурно-ориентированного языка. Компилятор реализован на языке Python с использованием библиотеки ply. Основная функция компилятора – трансляция модуля входного языка в эквивалентный модуль на языке ассемблера MIPS
Разработка компилятора подмножества процедурного языка в ассемблер состоит из следующих стадий:

- Разработка лексера
- Разработка парсера
- Разработка таблицы символов
- Генерация промежуточного кода
- Трансляция в целевой код

1 ПРОЕКТИРОВАНИЕ КОМПИЛЯТОРА

1.1 Основные требования

Разработка будет производиться в соответствии со следующими требованиями:

Требования к входному языку:

1. Должны присутствовать операторные скобки;
2. Должна игнорироваться индентация программы;
3. Должны поддерживаться комментарии любой длины;
4. Входная программа должна представлять собой единый модуль, но поддерживать вызов функций.

Требования к операторам:

1. Оператор присваивания.
2. Арифметические операторы (*, / , +, -, >, <, =).
3. Логические операторы (И, ИЛИ, НЕ).
4. Условный оператор (ЕСЛИ).
5. Оператор цикла (while).
6. Базовый вывод (строковой литерал, переменная).
7. Типы (целочисленный, вещественный).

Требования к выходному языку:

1. В ассемблере.

1.2 Лексический анализатор

Лексический анализатор преобразует входной поток символов в поток токенов.

Лексический анализатор реализован при помощи библиотеки ply.lex. Сам исходный язык представляет собой видоизменённое подмножество языка Pascal: вместо операторных скобок используются “start” и “stop”, объявление переменных происходит в блоке “var”.

Основная часть языка выполнена в виде зарезервированных слов: and, start, div, do, stop, for, func, if, mod, not, or и прочих.

Арифметика представлена в виде массива литералов в который так же включены точка, запятая, точка с запятой, двоеточие и скобки.

Так же в лексере содержатся правила определяющие комментарии в тексте программы, числа с плавающей точкой, строковые литералы, переход на новую строку и правило выбрасывающее ошибку при вводе некорректных символов.

1.3 Синтаксический анализатор

На вход синтаксический анализатор получает набор токенов из лексического анализатора. В самом парсере описаны правила их соединения - грамматика.

Грамматика языка, основная часть модуля:

```
'program :  program_heading semicolon block DOT'

'program_heading :  RESERVED_PROGRAM identifier'

'program_heading :  RESERVED_PROGRAM identifier LPAREN identifier_list RPAREN'

'identifier_list :  identifier_list comma identifier'

'identifier_list :  identifier'

'block :  declaration_part_list statement_part'

'block :  statement_part'
```

Рис. 1

Рассмотрим примеры использования правил выполнения операций:

Присваивание:

Для присваивания доступны числа, строковые литералы, объявленные переменные типов int, float и string.

```
'assignment_statement :  variable_access ASSIGNMENT expression'

'variable_access :  identifier'

'identifier :  IDENTIFIER | RESERVED_STRING'
```

Рис. 2

Арифметика:

Так как большинство операторов представлены в лексере в виде литералов, то и в правилах указаны так же литералы. В иных случаях указаны токены от зарезервированных слов.

```
'simple_expression :  simple_expression addop term'

'''addop :  '+' | '-' | RESERVED_OR '''

'term :  term mulop factor'

'''mulop :  '*' | '/' | RESERVED_DIV | RESERVED_MOD | RESERVED_AND '''

'expression :  simple_expression relop simple_expression'
```

Рис. 3

Условный оператор:

Блок обязательно начинается с зарезервированного слова *if*, далее идет само условие, зарезервированное слово *then* отделяющее оператор перехода и тело цикла

```
'open_if_statement :  RESERVED_IF boolean_expression RESERVED_THEN marker_for_branching statement'
```

Рис. 4

Функции:

Функция определяется после использования зарезервированного слова *func*. Далее определяются переменные, двоеточие и тип выходных данных. Вызов функции происходит путем ввода *func* и идентификатора функции, объявленного ранее.

```
'function_heading :  RESERVED_FUNCTION identifier COLON result_type'

'function_declaration :  function_heading semicolon function_block'

'function_identification :  RESERVED_FUNCTION identifier'
```

Рис. 5

Пример дерева для программы:

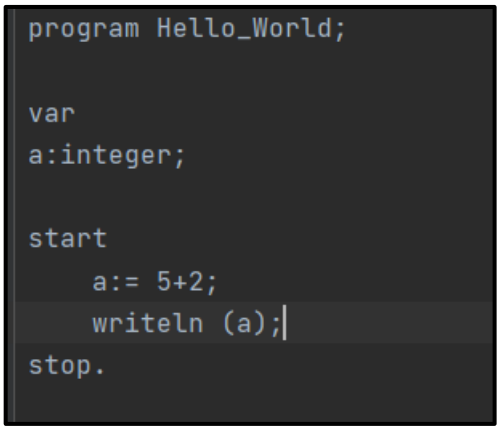

Рис. 6

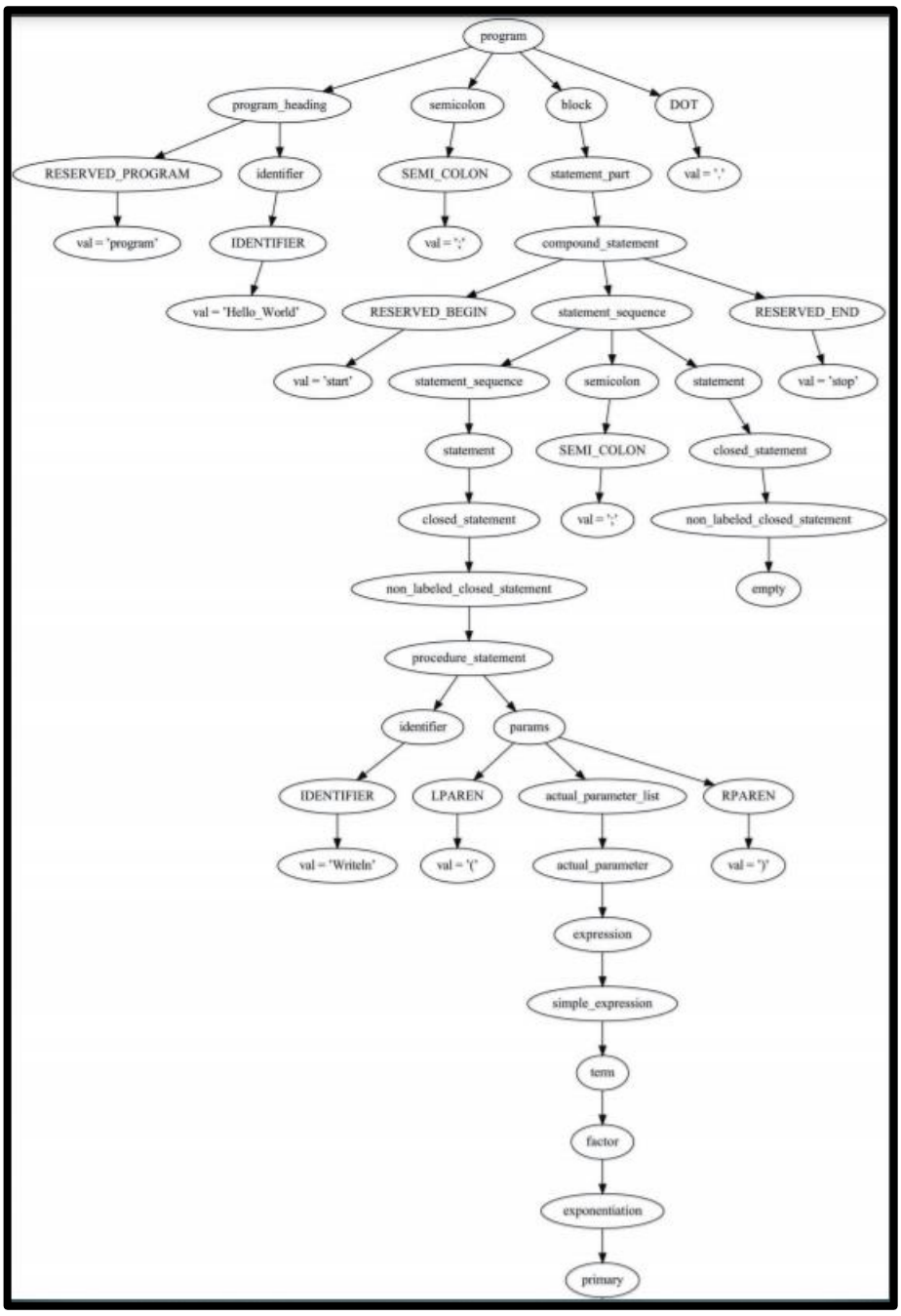
program
program_heading
semicolon
block
DOT
RESERVED_PROGRAM
identifier
SEMI_COLON
statement_part
val = '.'
val = 'program'
IDENTIFIER
val = ';'
compound_statement
val = 'Hello_World'
RESERVED_BEGIN
statement_sequence
RESERVED_END
val = 'start'
statement_sequence
semicolon
statement
val = 'stop'
statement
SEMI_COLON
closed_statement
closed_statement
val = ';'
non_labeled_closed_statement
non_labeled_closed_statement
empty
procedure_statement
identifier
params
IDENTIFIER
LPAREN
actual_parameter_list
RPAREN
val = 'Writeln'
val = '('
actual_parameter
val = ')'
expression
simple_expression
term
factor
exponentiation
primary

Рис. 7

1.4 Построение таблицы символов и генерация промежуточного представления

Таблица символов получает на вход от парсера информацию об объявленных переменных, функциях, их типах, уровнях вложенности и значениях.

Таблица символов представлена в виде хэш-таблицы. Таблица содержит в себе ключ и шесть значений. Ключом является временная переменная (t_i). Поля содержат в себе следующую информацию: offset - сдвиг в стеке, width – размер, type – тип, variable_name – имя переменной в программе, id_path – путь к переменной, name – имя временной переменной.

Полученные данные пересылаются в генератор промежуточного кода. Промежуточный код представляет собой трёхадресный код типа x:= y op z, который записывается в виде ‘dest(x), src1(y), src2(z), op’.

Поскольку трёхадресный код является представлением более близким к машинному, то и генерировать код ассемблера становится проще.

1.5 Трансляция в целевой код

На данном этапе промежуточный код подается на вход генератору машинного кода. Генератор прогоняет промежуточный код по множеству условий, таких как, например, наличие арифметический инструкций, вызова функций или вывода и при обнаружении таковых генерирует соответствующие инструкции на языке ассемблера MIPS.

Основные функции в генераторе:

- Addops – операции сложения, вычитания, присваивания, логическое И и ИЛИ
- Mulops – операции умножения, деления, целочисленного деления и остатка от деления
- Relops – операции сравнения
- Change_type – смена типов
- Jumps - переходы
- Output – вывод

2 ПРОВЕРКА НА СООТВЕТСТВИЕ ОСНОВНЫМ ТРЕБОВАНИЯМ

В задании на курсовую работу были предъявлены следующие требования к реализуемому компилятору:

Требования к входному языку:

1. Должны присутствовать операторные скобки;
2. Должна игнорироваться индентация программы;
3. Должны поддерживаться комментарии любой длины;
4. Входная программа должна представлять собой единый модуль, но поддерживать вызов функций.

Требования к операторам:

1. Оператор присваивания.
2. Арифметические операторы (*, / , +, -, >, <, =).
3. Логические операторы (И, ИЛИ, НЕ).
4. Условный оператор (ЕСЛИ).
5. Оператор цикла (while).
6. Базовый вывод (строковой литерал, переменная).
7. Типы (целочисленный, вещественный).

Требования к выходному языку:

1. В ассемблере.

Рассмотрим выполнение указанных выше требований на примерах работы компилятора:

Нахождение последовательности Фибоначчи:

```
program fibonacci;

func fib(n:integer): integer;
start
    if (n <= 2) then
    start
            fib := 1;
    stop;
    if (n > 2) then
    start
            fib := fib(n-1) + fib(n-2);
    stop;
stop;

var
    i:integer;
    e:integer;
    s:string;
    out : integer;

start
    for i := 1 to 20 do
    start
        out := fib(i);
        writeln(out);
    writeln(' ');
    stop
stop.
```

Рис. 8

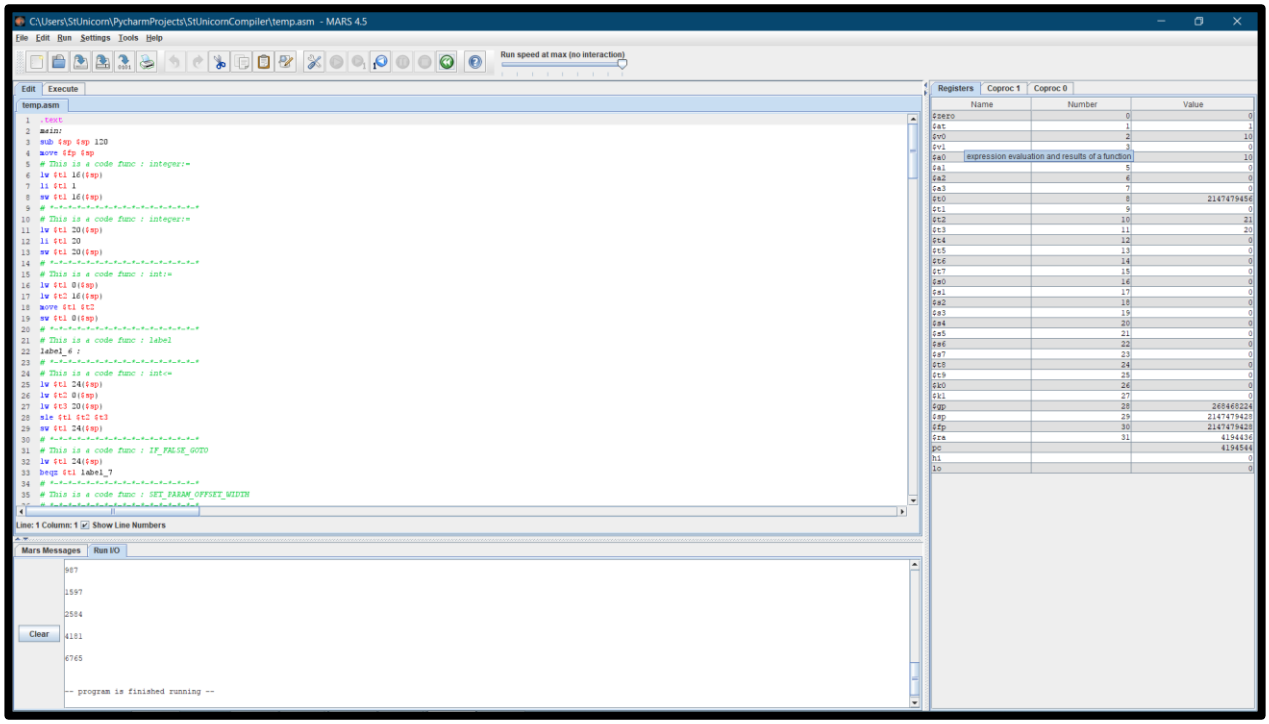

Рис. 9

Факториал:

```
program factorial;
var
n, i, s: integer;

start
  n := 12;
  s := 1;
  for i := 1 to n do
    start
        s := s * i;
    stop;
writeln(s)
stop.
```

Рис. 10

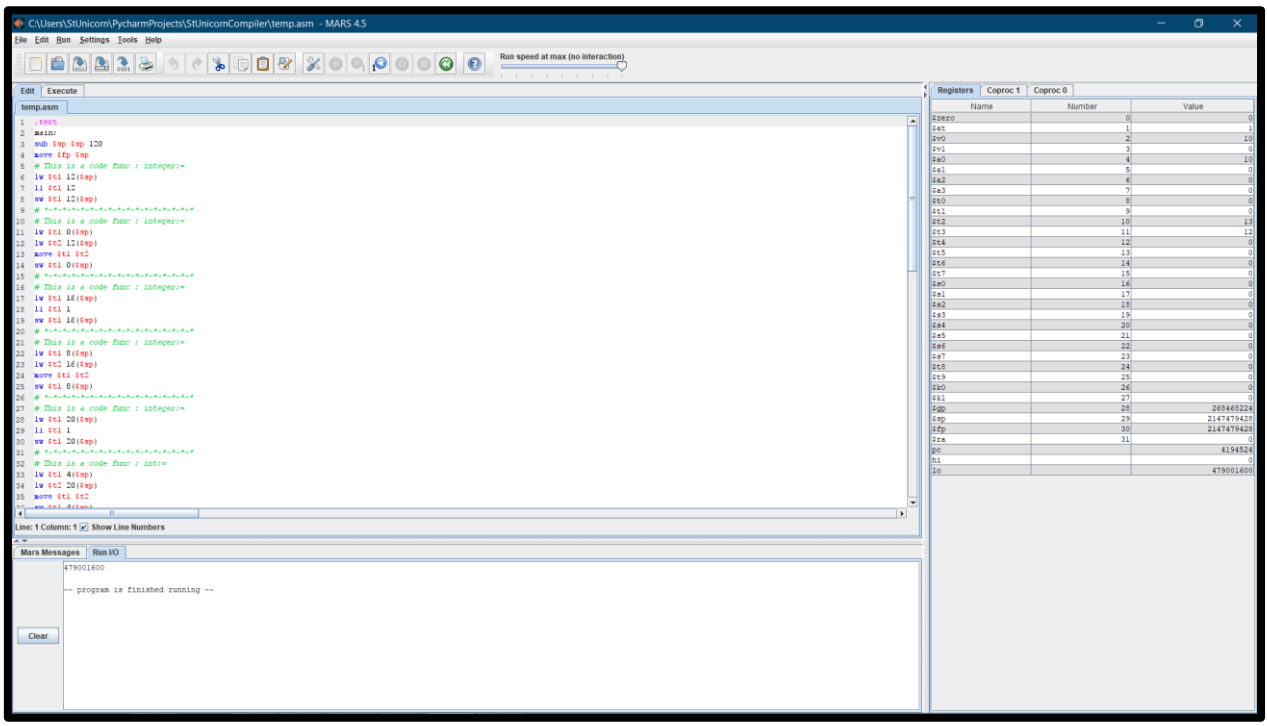

Рис. 11

Возведение в степень:

```
program pwr;

func power(numb,pow:integer):integer;
var i,res:integer;
start
    res:= 1;
    for i:= 1 to pow do
    start
        res:= res*numb;
    stop;
    power:= res;
stop;

var a,b,c:integer;
start
    a:= 4;
    b:= 3;
    c:= power(a,b);
    writeln(c);
stop.
```

Рис. 12

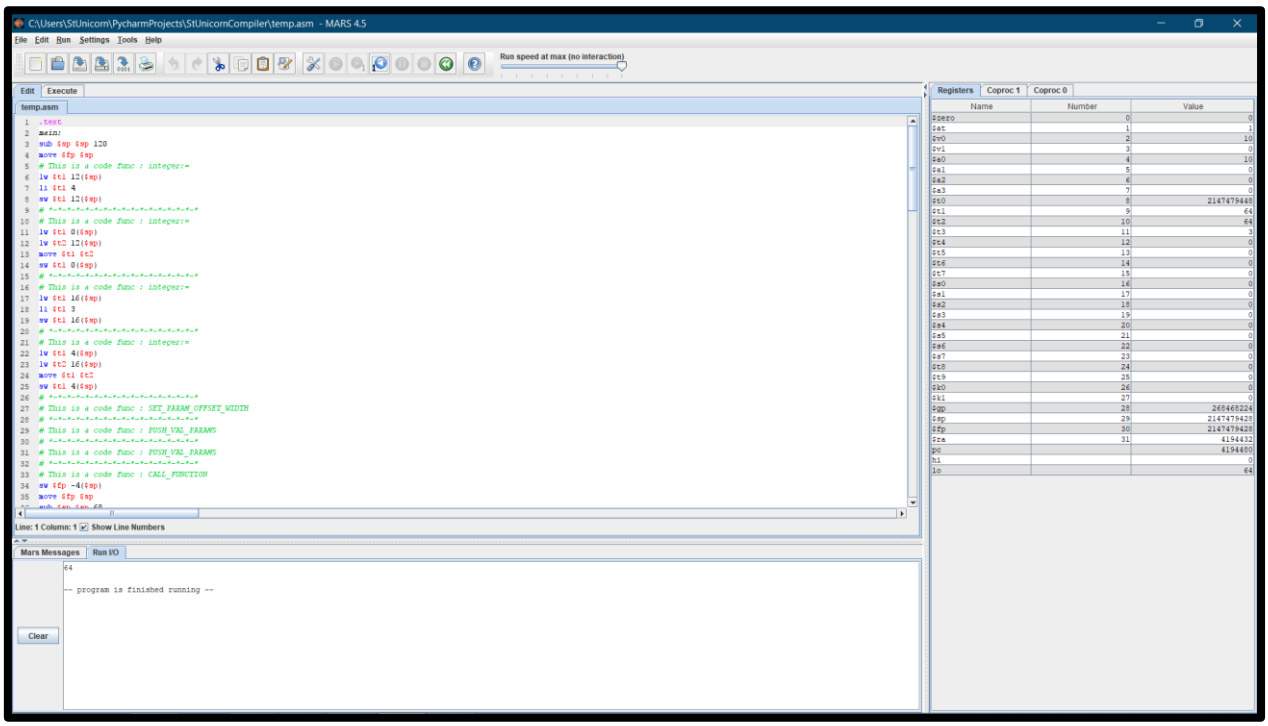
C:\Users\StUnicorn\PycharmProjects\StUnicornCompiler\temp.asm - MARS 4.5
File Edit Run Settings Tools Help
Run speed at max (no interaction)
Edit Execute
temp.asm
.text
main:
sub $sp $sp 120
move $fp $sp
This is a code func : integer:=
lw $t1 12($sp)
li $t1 4
sw $t1 12($sp)
--*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
This is a code func : integer:=
lw $t1 0($sp)
lw $t2 12($sp)
move $t1 $t2
sw $t1 0($sp)
--*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
This is a code func : integer:=
lw $t1 16($sp)
li $t1 3
sw $t1 16($sp)
--*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
This is a code func : integer:=
lw $t1 4($sp)
lw $t2 16($sp)
move $t1 $t2
sw $t1 4($sp)
--*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
This is a code func : SET_PARAM_OFFSET_WIDTH
--*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
This is a code func : PUSH_VAL_PARAMS
--*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
This is a code func : PUSH_VAL_PARAMS
--*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
This is a code func : CALL_FUNCTION
sw $fp -4($sp)
move $fp $sp
Line: 1 Column: 1 Show Line Numbers
Mars Messages Run I/O
64
-- program is finished running --
Clear
Registers Coproc 1 Coproc 0
Name Number Value
$zero 0 0
$at 1 1
$v0 2 10
$v1 3 0
$a0 4 10
$a1 5 0
$a2 6 0
$a3 7 0
$t0 8 2147479448
$t1 9 64
$t2 10 64
$t3 11 3
$t4 12 0
$t5 13 0
$t6 14 0
$t7 15 0
$s0 16 0
$s1 17 0
$s2 18 0
$s3 19 0
$s4 20 0
$s5 21 0
$s6 22 0
$s7 23 0
$t8 24 0
$t9 25 0
$k0 26 0
$k1 27 0
$gp 28 268468224
$sp 29 2147479420
$fp 30 2147479428
$ra 31 4194432
pc 4194460
hi 0
lo 64

Рис. 13