# Projeto do Compilador

Linguagem JSC

## Objetivo

O objetivo deste projeto é adquirir experiência na implementação de linguagens de programação através da construção de um compilador para uma simples linguagem chamada JSC, com sintaxe/semantica a partir de JavaScript e C.

## Avaliação

- **15%** Analisador Léxico
- **35%** Analisador Sintático/Semântico
- **50%** Gerador de Código

## A Linguagem

```
Program    = (DecVar | DecFunc)*

DecVar     = var ID ('=' Expr)? ';'

DecFunc    = function ID '(' ParamList? ')' Block
ParamList  = ID (',' ID)*

Block      = '{' DecVar* Stmt* '}'

Stmt       = Assign ';' |
             FuncCall ';' |
             if '(' Expr ')' Block (else Block)? |
             while '(' Expr ')' Block |
             return Expr? ';' |
             break ';' |
             continue ';'

Assign     = ID '=' Expr
FuncCall   = ID '(' ArgList? ')'
ArgList    = Expr (',' Expr)*

Expr       = Expr BinOp Expr |
```

```
        UnOp Expr |
        '(' Expr ')' |
        FuncCall |
        DEC |
        ID

BinOp     = '+' | '-' | '*' | '/' | '<' | '<=' |
        '>' | '>=' | '==' | '!=' | '&&' | '||'
UnOp      = '-'  | '!'
```

Embora não estejam refletidas explicitamente na gramática, todas as operações binárias na linguagem são associativas à esquerda e a precedência dos operadores é a mesma usada na linguagem C. Mais precisamente, há sete níveis de precedência dos operadores, listados abaixo do maior nível para a menor nível de precedência.

| **Operadores** | **Comentarios** |
|---|---|
| **-** | Negação unária --> maior precedência |
| **!** | Negação lógica unária (NOT) |
| *** /** | Multiplicação e divisão (inteiros) |
| **+ -** | Adição e subtração (inteiros) |
| **< <= >= >** | Relação lógica |
| **== !=** | Igualdade e desigualdade lógicas |
| **&&** | Conjunção lógica (AND) |
| **\|\|** | Disjunção lógica (OR) --> menor precedência |

Importante notar que da forma como a gramática está descrita acima, ela não é nem *LL (1)* nem *LR (1)*. Contudo, a gramática deve ser transformada (pelas técnicas vistas em sala de aula) para que seja possível a construção mais adequada do *analisador sintático* para a linguagem.

## Exemplo

```
// add.jsc - simple addition example
function add(x, y)
{
  return x + y; // add the two parameters
}
function main()
{
  var a;
  a = 3;
  print(add(a, 2));
}
```

O resultado correto da execução do código acima é o valor decimal 5, onde a função de entrada é definida pela função `main`.

## Considerações Léxicas

Há quatro classes de tokens na linguagem JSC:

```
ID    Identificador
DEC   Literal decimal (inteiro)
KEY   Palavra-chave
SYM   Símbolos léxicos
```

Os **identificadores** e **palavras-chave** devem começar com um caractere alfabético e podem conter caracteres alfanuméricos e o caractere de sublinhado `'_'`. Uma palavra-chave pode ser vista como um identificador especial que é reservado e nao pode ser usado para definir nomes de variáveis ou funções. As palavras-chave e os identificadores são sensíveis a maiúsculas e minúsculas, e todas as palavras-chave são representadas por minúsculas. Por exemplo, **if** é uma palavra-chave, mas **IF** pode ser um nome de variável; **foo** e **Foo** são dois exemplos de nomes diferentes na linguagem, podendo referir-se a duas variáveis distintas.

Note que palavras-chave e identificadores devem ser separados por espaço em branco, ou por algum token que não é nem uma palavra-chave nem um identificador. Por exemplo, **ifbreak** é um único identificador, e não duas palavras-chave distintas. Se uma sequência de caracteres começa com um caractere alfabético, então ele deve representar a sequência mais longa de caracteres alfanuméricos e sublinhados, formando um único token (palavra-chave ou identificador).

Os **símbolos** podem ser divididos em três tipos: 1) operadores de agrupamento: parênteses, chaves, colchetes, atribuição (sinal de igual), vírgulas e ponto-e-vírgula, 2) operadores binários (BINOP) e 3) operadores unários (UNOP). Há uma variedade de operadores binários e unários na linguagem, incluindo tanto os operadores aritméticos (por exemplo, mais e menos) e operadores lógicos/relacionais (por exemplo, ou/e booleano e menor-que). Abaixo está a lista de todos os **símbolos léxicos** da linguagem:

```
( { [ ] } ) , ; = + - * / < > <= >= == != && || !
```

As seguintes **palavras-chave** são *reservadas* na linguagem:

```
var function if else while return break continue
```

Os comentários devem ser iniciados por "//" e vão até o final da linha. Um token especial "Whitespace" pode aparecer entre quaisquer tokens, e consiste em um ou mais espaços em branco, tabs (\t), e quebra de linhas (\n). Comentários e espaços em branco devem ser descartados durante a fase de análise léxica do compilador.

# Semântica

Um programa escrito na linguagem consiste em definições de variáveis e funções, respeitando escopos léxicos correspondentes.

## Variáveis

A linguagem permite somente o uso de um tipo de variável: inteiro com sinal (32-bit). Variáveis, consideradas *globais*, podem ser declaradas fora de funções e são visíveis por todo código. Variáveis

consideradas *globais*, podem ser declaradas fora de funções e são visíveis por todo código. Variáveis declaradas dentro de funções ou blocos, consideradas *locais*, somente são visíveis dentro daquela função ou bloco (escopo léxico). Se uma variável for referenciada antes de uma atribuição, considera-se que a variável foi inicializada com valor zero (0). Variáveis devem ser declaradas antes de serem usadas.

### Funções

Funções podem não retornar *explicitamente* valor algum ou retornam um único valor, mas podem receber um número ilimitado de argumentos. Todo código escrito na linguagem deve conter uma função chamada `main` (sem argumentos) que representa o início da execução do programa. Funções que não retornam valores explicitamente devem retornar valor *zero (0)*.

### Escopo léxico

Há dois escopos válidos em um programa Decaf: *global* e *local*. O escopo global consiste em nomes de funções e variáveis declaradas fora de funções. O escopo local consiste em nomes de variáveis e parâmetros formais declarados em uma função. Escopos locais adicionais podem existem dentro de cada bloco no código, como após as construções `if` ou `while`, ou em qualquer parte do código onde há um novo bloco. Nenhum nome de identificador pode ser definido mais de uma vez no mesmo escopo. Assim, nomes de variáveis e funções devem ser distintos no escopo global, e nomes de variáveis locais e nomes de parâmetros formais devem ser distintos em cada escopo local.

## Entrada/Saída

A linguagem prevê a função **`print`** `'('` *`Expr`* `')'` que não é definida na gramática da linguagem, e sim parte do ambiente de execução da linguagem. A função exibe o valor avaliado de uma expressão na saída padrão seguido de uma quebra de linha (\n). Para isso, será usada uma *syscall* no MIPS (detalhes na seção abaixo).

## Notas sobre Implementação

O compilador implementado deve gerar código correto (não necessariamente otimizado) para o processador MIPS (32 bits); detalhes da arquitetura MIPS aqui e do conjunto de instruções aqui.

**Requisitos da Implementação:**

- **Deve** ser implementado em C/C++
- **Deve** rodar no ambiente Linux/Unix
- *Pode* opcionalmente usar ferramentas automáticas (Lex/Flex & Yacc/Bison)

O programa do compilador deve receber dois parâmetros/argumentos (argv). O primeiro argumento será o arquivo do código fonte como entrada (e.g., add.jsc) e o segundo argumento será o arquivo de saída (e.g., out.asm) onde será gravado o código em assembly MIPS; por exemplo:

```
$ ./compilador add.jsc out.asm
```

Para executar e avaliar os resultados gerados pelos programas da linguagem alvo (MIPS assembly) iremos

usar o simulador SPIM (detalhes no manual aqui). O SPIM (MIPS32 Simulator) pode ser baixado aqui. Veja aqui detalhes das system calls disponíveis no simulador SPIM.

Para saber mais sobre geradores automáticos de analisadores léxico e sintático, leia os seguintes ponteiros: Flex in a Nutshell e Introduction to Bison.

# Formato de Arquivos

## Analisador Léxico

- **Como executar (dois parametros: entrada e saída)**

```
$ ./lexico add.jsc out.lex
```

- **Exemplo de Arquivo de Entrada:**

```
function main()
{
  var a;
  a = 4 + 5;
  return a;
}
```

- **Exemplo de Arquivo de Saída (quaisquer espaços em branco serão ignorados):**

```
KEY  "function"
ID   "main"
SYM  "("
SYM  ")"
SYM  "{"
KEY  "var"
ID   "a"
SYM  ";"
ID   "a"
SYM  "="
DEC  "4"
SYM  "+"
DEC  "5"
SYM  ";"
KEY  "return"
ID   "a"
SYM  ";"
SYM  "}"
```

## Analisador Sintático

- **Como executar (dois parametros: entrada e saída)**

```
$ ./lexico add.jsc out.syn
```

Há várias possíveis árvores corretas que podem ser geradas para uma entrada de um programa. Assim, é importante ter um formato único para representação da árvore sintática (abstrata) que contenha um número mínimo de elementos/nós e seja independente de qualquer implementação específica. A saída deve estar na notação estilo LISP:

```
[operador [operando_1] ... [operando_N]]
```

Recursivamente, cada operando pode ser definido por um outro operador; por exemplo,

```
[op_1 [op_2 [a] [b]] [c]]
```

onde `"op_1"` possui dois operandos: `"[op_2 [a] [b]]"` e `"[c]"`, e operador `"op_2"` possui dois operandos: `"[a]"` e `"[b]"`.

Como há varias formas de gerar a saída AST para um mesmo código de entrada, devemos um formato uniforme de saída da AST que seja independente de qualquer implementação do analisador sintático. Abaixo temos o exemplo dos elementos/nós (e seus respetivos níveis) usados para criar a AST:

- `Program`
  - `DecVar`
    - `ID`
    - *Expr* ...
  - `DecFunc`
    - `ID`
    - `ParamList`
    - `Block`
      - ...
- **Exemplo de codigo fonte:**

```
function main() { var a = 3 + 5; var b; }
```

- **Exemplo da arvore AST:**

```
[program [decfunc [main] [paramlist] [block [decvar [a] [+ [3] [5] ] ][decvar [b]] ] ]
```

- **Outro exemplo de Arquivo de Entrada**

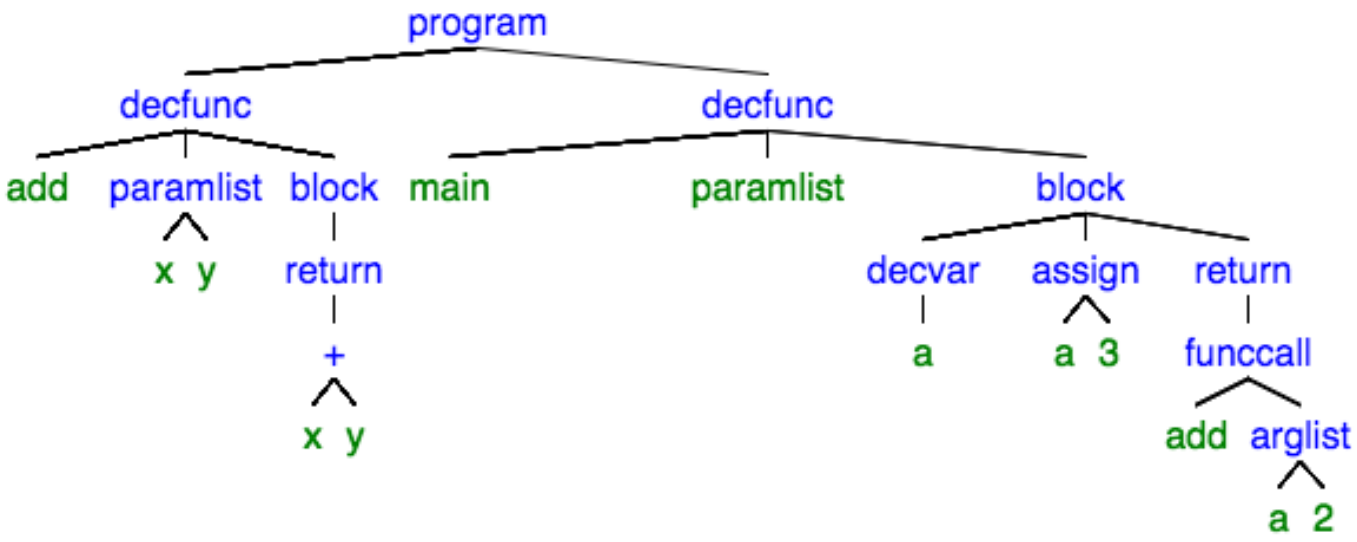
```
function add(x, y)
{
   return x + y;
}

function main()
{
   var a;
   a = 3;
```

```
    return add(a, 2);
}
```

- **Exemplo de Arquivo de Saída (**AST em "labelled bracket notation")

```
[program
  [decfunc
    [add]
    [paramlist [x] [y]]
    [block
      [return
        [+ [x] [y]]
      ]
    ]
  ]
  [decfunc
    [main]
    [paramlist ]
    [block
      [decvar [a]]
      [assign [a] [3]]
      [return
        [funccall
          [add]
          [arglist [a] [2]]
        ]
      ]
    ]
  ]
]
```

A AST pode ser visualizada abaixo (gerada por este site aqui):




## Gerador de código

- **Como executar (dois parametros: entrada e saída)**

```
$ ./codegen add.jsc add.asm
```

- **Exemplo de Arquivo de Entrada:**

```
function main() { print(4+5); }
```

- **Exemplo de Arquivo de Saida (MIPS assembly):**

```
.data

.text

_f_print:
  lw $a0, 4($sp)
  li $v0, 1
  syscall
  li $v0, 11
  li $a0, 0x0a
  syscall
  addiu $sp, $sp, 4
  lw $fp, 4($sp)
  addiu $sp, $sp, 4
  j $ra

_f_main:
  move $fp, $sp
  sw $ra, 0($sp)
  addiu $sp, $sp, -4
  sw $fp, 0($sp)
  addiu $sp, $sp, -4
  li $a0, 4
  sw $a0, 0($sp)
  addiu $sp, $sp, -4
  li $a0, 5
  lw $t1, 4($sp)
  addiu $sp, $sp, 4
  add $a0, $t1, $a0
  sw $a0, 0($sp)
  addiu $sp, $sp, -4
  jal _f_print
  lw $ra, 4($sp)
  addiu $sp, $sp, 4
  lw $fp, 4($sp)
  addiu $sp, $sp, 4
  j $ra

main:
  sw $fp, 0($sp)
  addiu $sp, $sp, -4
```

```
jal _f_main
li $v0, 10
syscall
```

- **Execução com simulador MIPS (spim):**

Importante notar que a avaliação do compilador será a saída produzida pelo SPIM, e *não* o código de montagem (assembly) produzido.

```
$ spim -file add.asm
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spim/exceptions.s
9
```