

# siC: Uma linguagem baseada em C incluindo fila como tipo primitivo

Gabriella de Oliveira Esteves, 110118995

<sup>1</sup>Departamento de Ciência da Computação - Universidade de Brasília

## Objetivo

Este trabalho visa projetar e construir uma nova linguagem chamada de siC - Structure in C, baseada na linguagem C. O siC acrescenta a estrutura de dados fila como tipo de dado primitivo e, para manipulá-la, adiciona certas operações próprias para tal.

## Introdução

Um compilador é um programa que recebe como entrada um código fonte e o traduz para um programa equivalente em outra linguagem [1]. Ele pode ser dividido em sete fases, ilustrado na Figura 1.

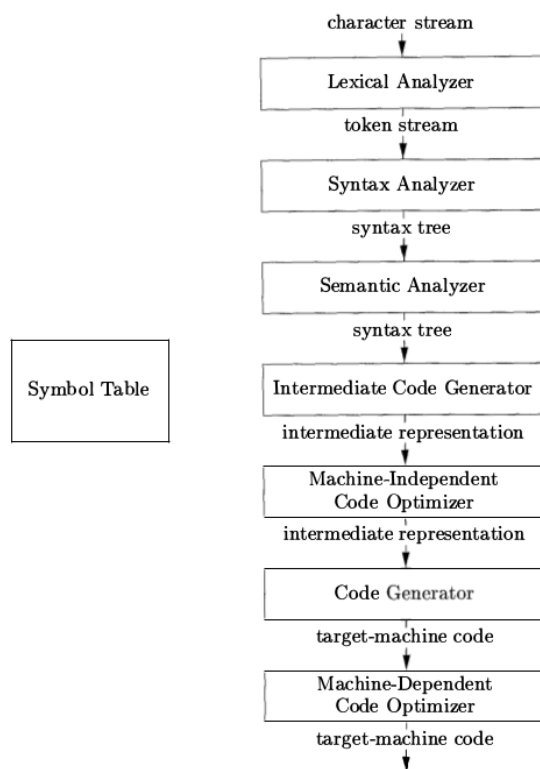


Figura 1. Fases de um compilador

- 1 **Analisador Léxico:** Lê o código fonte e atribui significado à cada sequência de caracteres, agora chamados lexemas. Cada lexema é mapeado para um token, que por sua vez é um par de nome (símbolo abstrato) e atributo (ponteiro para tabela de símbolos);
- 2 **Analisador Sintático:** Constrói uma representação gramatical dos tokens em forma de árvore;

- 3 **Analizador Semântico:** Utiliza a árvore sintática juntamente com a tabela de símbolos para verificar se a consistência semântica é mantida de acordo com a definição da linguagem.
- 4 **Gerador de Código Intermediário:** Converte árvore sintática anotada em código intermediário, com linguagem parecida com assembly e que possui apenas três operadores por linha de código. Nesse sentido, quebra-se estruturas complexas em estruturas mais simples, nesta fase.
- 5, 6 **Otimizador de Código Independente/Dependente de Máquina:** Procura aprimorar o código intermediário com o objetivo de melhorar o código-alvo de alguma forma: o deixando mais rápido, mais curto, consumindo menos energia, etc.
- 7 **Gerador de Código:** Converte o código intermediário no código-alvo, buscando atribuir os registradores às variáveis da maneira ótima.

O foco do projeto será nas fase 1, 2, 3 e 4, porém a princípio serão apresentadas apenas a descrição da linguagem siC, uma breve descrição de sua semântica e o analisador léxico. Como a fila é uma das estruturas de dados mais básicas, é possível dizer que siC se destina a inúmeras áreas de Ciência da Computação, como, por exemplo, sistemas operacionais, onde ela é usada para organizar prioridades dos processos.

Dois grandes motivos sustentam a escolha do tema deste projeto. Primeiro, uma vez que a fila faz parte dos tipos primitivos de uma linguagem, haverá menos manipulação de ponteiros na mesma, portanto erros envolvendo-os são menos prováveis de ocorrer. Segundo, a linguagem siC é mais alto-nível que C devido à abstração desta estrutura de dados básicas, e, de maneira geral, pode ser mais *user-friendly*. Nesse sentido, o usuário (da linguagem) leigo deverá entender como a estrutura funciona, bem como suas vantagens/desvantagens e usabilidade; porém a implementação de cada uma estará a cargo da própria siC.

## Gramática

A seguir será apresentada a gramática da linguagem siC, baseada em C [2]. Alguns comentários são feitos ao longo da gramática para facilitar o entendimento das variáveis e nomenclatura utilizada. As palavras reservadas da linguagem são representadas aqui como *tokens*. As variáveis e constantes são representadas como *identifiers*, que por sua vez é uma expressão regular, e a única diferença entre este e *identifier\_struct* é que o segundo tem acesso ao início da fila caso este seja o tipo do *identifier*.

Antes de apresentá-la, segue abaixo algumas alterações e correções da versão passada deste arquivo:

- Será implementado apenas a fila como tipo primitivo. Se fosse mantida a primeira proposta de incluir também o tipo pilha, talvez não seria possível terminar o projeto no prazo previsto;
- O tipo booleano deixará de existir em siC, porém o tipo float será incluído;
- Agora, além da variável *argument*, também existe a *arguments*, que permite a definição de zero ou mais argumentos em uma função;
- Na primeira versão da gramática o *statement* estava envolvido entre chaves no IF e no WHILE, enquanto nesta versão as chaves não são mais obrigatórias, porém a regra *statement*  $\rightarrow$  { *statement* } foi adicionada para criação de blocos;

- Expressões matemáticas são agora da forma *identifier* → *assignment\_expression* ao invés de *identifier* → *factor* para maior legibilidade;
- As aspas que delimitavam os símbolos como chaves e parênteses foram retiradas para aumentar também a legibilidade;
- Foram adicionados as operações de comparação < e >;
- O símbolo \$ foi adicionado na variável *letra*, pois em C ele pode compor um identificador;
- A variável *caractere* foi adicionada para representar o valor de um char, que só poderá ser ou uma letra ou um dígito;
- A variável inicial será agora *program*, que possibilitará a criação de uma ou mais funções no código fonte;
- Como existe agora chamada de função, foi criado mais um *statement* para tal, *identifier(identifiers)*; que exigiu a criação da nova variável *identifiers* para a passagem de parâmetros. O retorno das funções pode ser atribuído para uma variável, portanto também foi criada a regra *identifier = identifier(identifiers)*;

Segue abaixo a gramática proposta cuja variável inicial é *program*, com as alterações acima em vermelho e as características diferenciais da linguagem siC em negrito.

token: WHILE, IF, ELSE, RETURN

token: QUEUE, FIRST, VOID, **FLOAT**, INT, CHAR

**program**

→ **program function**  
| **function**

**function**

→ **argument** ( arguments ) { statement RETURN identifier ; }

**identifiers**

→ **identifiers, identifier**  
| **identifier**  
| **ε**

**identifier**

→ letra(letra | dígito)\*

**caractere**

→ letra | dígito

**letra**

→ a | b | ... | z | A | B | ... | Z | \$

**dígito**

→ 0 | 1 | ... | 9

**identifier\_struct**

→ **identifier**  
| **identifier . FIRST**

**type\_struct**

→ **type\_simple**  
| **type\_queue**

```

type_simple
→ VOID | FLOAT | INT | CHAR

```

Existe um novo tipo de dado, *QUEUE*, que será composto por tipos simples de dados apenas (ou seja, não será possível criar uma variável do tipo fila em que seus elementos também são filas). Caso a variável seja do tipo fila, ela poderá obter o primeiro elemento através do comando "identifier.FIRST".

```

type_queue
→ QUEUE < type_simple >

```

```

arguments
→ arguments , argument
| argument
| ε

```

```

argument
→ type_struct identifier

```

A seguir serão descritas quatro estruturas básicas da linguagem siC: comando com repetição, condicional, expressões matemáticas e expressões com pilhas e filas. A última contempla as operações de adicionar elemento no topo da pilha ou no fim da fila, "+", e remover do topo ou do início da fila, "-", onde o valor do elemento retirado é armazenado no último operando da expressão.

```

% =====
% =====
% =====
REPENSAR
REPENSAR
REPENSAR
REPENSAR

```

```

% =====
% =====
% =====

```

```

matched_statements
→ matched_statements matched_statement
| ε

```

```

statement
→ matched_statement
| open_statement

```

```

matched_statement
→ IF ( compare_expression ) matched_statement ELSE matched_statement
| identifier ( identifiers ) ;
| identifier = identifier ( identifiers ) ;
| WHILE ( compare_expression ) matched_statement
| identifier = assignment_expression ;
| identifier_struct_expression

```

```

    | { matched_statements }

open_statement
    → IF ( compare_expression ) statement
    | IF ( compare_expression ) matched_statement ELSE open_statement

% =====
% =====
% =====
REPENSAR
REPENSAR
REPENSAR
REPENSAR

% =====
% =====
% =====

statements
    → statements statement
    | ε

statement
    → argument ';'
    | identifier ( identifiers ) ;
    | identifier = identifier ( identifiers ) ;
    | IF ( compare_expression ) statement
    | IF ( compare_expression ) statement ELSE statement
    | WHILE ( compare_expression ) statement
    | identifier = assignment_expression ;
    | identifier_struct_expression
    | { statements }

compare_expression
    → identifier_struct compare_assignment identifier_struct

compare_assignment
    → == | != | <= | >= | < | >

assignment_expression
    → assignment_expression + term
    | assignment_expression - term
    | term

term
    → term * factor
    | term / factor
    | factor

factor
    → identifier_struct
    | ' caractere '
    | ( assignment_expression )

identifier_struct_expression

```

```
→ identifier = identifier + identifier ;  
| identifier = identifier - identifier ;
```

## Analizador Léxico

A principal tarefa do analisador léxico é examinar cada elemento do código fonte (variáveis, símbolos, números, etc), reconhecê-los com base em certos *tokens* e classificá-los em grupos de lexemas. Além disso, ele pode realizar outras funções como eliminar espaços em brancos, tabulações, quebras de linhas e comentários; armazenar e acompanhar os números da linha e coluna corrente no momento de sua execução; Informar mensagens de erro ou avisos de prevenção de erro diretamente ao usuário da linguagem [1].

Existem três termos distintos bastante relacionados com analisador léxico. O primeiro, já citado anteriormente, é o *token*, um par onde o primeiro elemento é o *token name* (símbolo abstrato que representa um tipo de unidade léxica, como a palavra chave "while", por exemplo) e onde o segundo elemento é um atributo (informação adicional e opcional sobre o *token*). O segundo termo é o *pattern*, uma descrição, em forma de expressão regular, do um lexema de um token. No caso de um número inteiro, por exemplo, o *pattern* seria uma sequência de um ou mais dígitos de 0 à 9. Por fim, o termo lexema significa uma sequência de caracteres no programa fonte que correspondem com o *pattern* de um lexema específico, ou seja, cada lexema é uma instância de um token. No caso de existirem mais de uma correspondência, o lexema será a instância do *token* cujo *pattern* aparece primeiro no arquivo .lex.

Esta seção apresenta o analisador léxico FLEX, utilizado neste trabalho, além de toda a descrição do arquivo .lex contruído a partir da gramática descrita no capítulo anterior.

### FLEX: The Fast Lexical Analyzer

Flex é uma ferramenta que gera um programa, chamado de *scanner*, cuja função é identificar *patterns* no código fonte. Ele recebe como entrada um arquivo de entrada especificados pelo usuário que serão reconhecidos a partir de expressões regulares mescladas com código em C (chamadas de descrição) no arquivo .lex [3]. Com o comando *Flex nome\_do\_arquivo.lex*, um código fonte chamado *lex.yy.c* é criado e nele existe uma função chamada *yylex()*, a qual realiza de fato as operações do *scanner*. Esse código, ao ser compilado corretamente com a flag *-lfl* da biblioteca do flex, gera um arquivo objeto executável que recebe uma entrada qualquer e gera uma saída que depende do código em C que foi escrito no .lex (imprimir o lexema identificado, contabilizar o número de linhas, imprimir mensagem de erro léxico, etc).

Neste projeto foram utilizadas duas variáveis globais muito úteis: *yytext* e *yyin*. A primeira contém o lexema que foi reconhecido como *token*. Esta variável é modificada sempre que um novo lexema é identificado no código fonte. A segunda define como a entrada será lida, que pode ser tanto pela entrada padrão quanto por arquivo.

### Arquivo lex

O código fonte de extensão .lex é composto por três partes: definições, regras e código em C do usuário. Na seção de definições é onde são declarados nomes para certas

	<i>Pattern</i>	<i>Ação</i>
1	<code>\n</code>	Identifica uma quebra de linha e incrementa variável <i>lines</i> para contagem de linhas
2	<code>[ \t]+</code>	Identifica um ou mais espaço ou tabulação
3	<code>"/"[\^\\n]*</code>	Ignora tudo a frente do comentário de uma linha <code>"/"</code> exceto quebra de linha
4	<code>{dígito}+{letra}+{dígito}*</code>	Gera o erro identificado na linha <i>lines</i> : Sufixo inválido no número inteiro
5	<code>{dígito}+."{dígito}*{letra}+{dígito}*</code>	Gera o erro identificado na linha <i>lines</i> : Sufixo inválido no número float
6	<code>{dígito}+."{dígito}*</code>	Identifica números float
7	<code>{dígito}+</code>	Identifica números inteiros
8	<code>""({letra} {dígito})""</code>	Identifica valor para uma variável do tipo char, que pode ser uma letra ou dígito
9	<code>""</code>	Gera o erro identificado na linha <i>lines</i> : Constante de caractere vazia
10	<code>{comparison}</code>	Identifica símbolos de comparação entre dois elementos
11	<code>{mark}</code>	Identifica pontuação e delimitadores de blocos e valores de char
12	<code>{operator}</code>	Identifica operadores matemáticos
13	<code>(?i:"VOID")</code>	Identifica palavra chave tipo void com as letras em caixa-alta ou caixa-alta.
14	<code>(?i:"FLOAT")</code>	Identifica palavra chave tipo float com as letras em caixa-alta ou caixa-alta
15	<code>(?i:"INT")</code>	Identifica palavra chave tipo inteiro com as letras em caixa-alta ou caixa-alta
16	<code>(?i:"CHAR")</code>	Identifica palavra chave tipo char com as letras em caixa-alta ou caixa-alta
17	<code>(?i:"QUEUE")</code>	Identifica palavra chave tipo fila com as letras em caixa-alta ou caixa-alta
18	<code>(?i:"FIRST")</code>	Identifica palavra chave que representa primeiro elemento da fila com as letras em caixa-alta ou caixa-alta
19	<code>(?i:"IF")</code>	Identifica palavra chave condicional if com as letras em caixa-alta ou caixa-alta
20	<code>(?i:"ELSE")</code>	Identifica palavra chave condicional else com as letras em caixa-alta ou caixa-alta
21	<code>(?i:"WHILE")</code>	Identifica palavra chave do laço while com as letras em caixa-alta ou caixa-alta
22	<code>(?i:"RETURN")</code>	Identifica palavra chave return, de retorno de função, com as letras em caixa-alta ou caixa-alta
23	<code>{id}</code>	Reconhece um identificador cuja regra é: não é permitido conter símbolos além de \$, letras e dígitos e não é permitido começar a palavra com dígito
24	<code>.</code>	Gera o erro identificado na linha <i>lines</i> : Token desconhecido

**Tabela 1. Tabela de regras**

	Nome	Definição
1	digito	[0-9]
2	letra	[a-zA-Z\$]
3	comparison	==   !=   <=   >=   <   >
4	mark	.   ;   ,   '   {   }   (   )
5	operator	+   -   *   /
6	id	[a-zA-Z\$][a-zA-Z\$0-9]*

**Tabela 2. Tabela de definições**

expressões regulares, para facilitar a escrita das regras na próxima seção. Neste projeto foram feitas seis definições, apresentadas na Tabela 2. A primeira representa um dígito apenas, de 0 à 9. A segunda é uma letra maiúscula ou minúscula e o símbolo \$, que em C pode compor o nome de um identificador. A terceira representa os símbolos de comparação entre dois números elementos (que serão do tipo char, inteiro e float). A quarta representa delimitadores do código siC, para finalizar um comanto, definir um escopo, etc. A quinta definição apresenta as quatro operações matemáticas básicas. A sexta e última apresenta uma expressão regular que especifica o formato de um identificador: ele deve começar com uma letra ou \$ e pode terminar com letras, \$s ou dígitos.

A segunda parte do código lex é composto pelas regras que são um par de *pattern* e ação que devem estar na mesma linha. A Tabela 1 apresenta cada regra utilizada no projeto. No código fonte, todos os elementos da entrada que são identificados são imprimidos, exceto quebra de linha, espaços, tabulações e comentários, que são ignorados. Existem três variáveis contadoras que são utilizadas nas ações: *lines*, que começa com um e é incrementada sempre que uma quebra de linha é reconhecida, e *errors*, que começa com zero e é incrementada sempre que um erro léxico é encontrado. Ao longo da execução são imprimidos as descrições dos erros léxicos na tela e, ao final, o número total de erros.

A ordem em que as regras estão é importante para o funcionamento correto do programa, pois se existir mais de uma correspondência de *pattern* para um elemento, ele será identificado pela regra que aparecer primeiro. Neste projeto, as regras das keywords deve vir antes das regras dos identificadores, pois assim, se a seguinte entrada *int x = 2;* for lida, por exemplo, o elemento *int* será reconhecido como identificador e também tipo inteiro, porém a regra de keyword deverá identificá-lo.

Os *patterns* contidos entre chaves foram definidos na seção de descrição. Em relação aos demais, segue abaixo a descrição de algumas expressões regulares.

- [^\\n ] : Reconhece tudo exceto espaço e quebra de linha (Exemplo: regra 3);
- {digito} : Reconhece apenas um dígito (Exemplo: regra 8);
- {digito}\* : Reconhece zero ou mais dígitos (Exemplo: regra 6);
- {digito}+ : Reconhece um ou mais dígitos (Exemplo: regra 5);
- “abc” : Reconhece a sequência de caracteres "abc"(Exemplo: regra 9);
- “a” | “b” : Reconhece o caractere "a" ou o "b"(Exemplo: regra 8);
- (?:“AB”) : Reconhece as sequências "AB", "Ab", "aB", "ab"(Exemplo: regra 13);
- . : Reconhece qualquer elemento (Exemplo: regra 24).



A terceira parte do código lex é composta por código em C, que define como será lida a entrada (por arquivo ou pela entrada padrão) e define também as ações tomadas por cada *pattern*.

## Erros Léxicos

Foram reconhecidos quatro erros léxicos em siC. Caso exista um elemento que comece com dígitos e termine com letras, o usuário será informado de que o sufixo de letras é inválido para um tipo inteiro (regra 4). Caso exista um elemento que comece com dígitos, tenha depois um ponto, e termine com letras, o usuário será informado de que o sufixo de letras é inválido para um tipo float (regra 5). Caso exista na entrada duas aspas simples, uma seguida da outra, o programa entende que entre eles deveria existir algum caractere que seria o valor que algum char, portanto o usuário será informado de que a constante de caractere está vazia (regra 9). Por fim, caso exista algum elemento não identificado na entrada, o usuário será informado (regra 24).

Para testar o código foram criados dois arquivos de extensão .sic, um de acordo com as normas especificadas neste projeto (teste\_correto.sic), outro com todos os quatro tipos de erros léxicos (teste\_errado.sic). Cinco erros léxicos são reportados neste último arquivo:

- 1 **ERROR on line 4 : Invalid suffix on integer “0i0”**  
Erro de variável começando com dígito, ou número inteiro contendo algum caractere;
- 2 **ERROR on line 9 : Invalid suffix on floating “0.0a0”**  
Erro de número float contendo algum caractere;
- 3 **ERROR on line 16 : Unknown token ‘!’**  
Erro de caractere desconhecido;
- 4 **ERROR on line 17 : Empty character constant “ ”**  
Nesta linha ocorreu a inserção de um caractere vazio na pilha q;
- 5 **ERROR on line 21 : Unknown token ‘@’**  
Outro erro de caractere desconhecido.

## Dificuldades enfrentadas

Nesta fase do projeto, as principais dificuldades foram construir as expressões regulares que formam os *patterns*, bem como ordená-las de forma que o analisador respeite as regras de precedência de reconhecimento dos padrões. Além disso, foi um desafio procurar por erros léxicos, uma vez que os principais e mais conhecidos são sintáticos.

## Analisador Semântico

A análise semântica utiliza da árvore sintática para checar a consistência da linguagem. Uma de suas obrigações mais importantes é a checagem de tipo. No caso do siC, existem várias restrições a serem consideradas:

- Para adicionar um elemento A de tipo simples (char, int ou float) no fim da fila de um elemento struct B, a atribuição deve ser do tipo  $B = B + A$ , onde A deverá ter tipo compatível com o de B, ou seja, se B for fila de inteiros, A deve ser um inteiro;

- Para remover um elemento A de tipo simples (char, int ou float) do início da fila de um elemento struct B, a atribuição deve ser do tipo  $B = B - A$ , onde A deverá ter tipo compatível com o de B, ou seja, se B for fila de inteiros, A deve ser um inteiro;
- Nenhuma operação matemática (*assignment\_expression*) pode conter um identificador B do tipo fila, apenas seu início, ou seja, B.FIRST.

Um exemplo de código em siC é apresentado a seguir. O programa adiciona três elementos numa fila de inteiros e depois eles são somados um a um e armazenados na variável *sum*. Ao final, a variável *lixo*, recém retirada da fila, é adicionada à *sum*. Nesse sentido, o resultado final de *sum* deve ser 7.

```

1 VOID main () {
2     QUEUE<INT> q;
3     INT sum, INT lixo;
4
5     q = q + 0;
6     q = q + 1;
7     q = q + 2;
8     q = q + 3;
9     sum = 0;
10
11     WHILE (q.FIRST != 0) {
12         sum = (sum + q.FIRST);
13         q = q - lixo;
14     }
15     sum = sum + lixo;
16
17     RETURN 0;
18 }
```

## Referências

- [1] A. V. Abo, M. S. Lam, R. Sethi, J. D. Ullman, *Compilers - Principles, Techniques and Tools* 2nd ed. 1986
- [2] ANSI C Yacc grammar, <http://www.quut.com/c/ANSI-C-grammar-y.html>, 18 12 2012.
- [3] Flex: The Fast Lexical Analyser, <http://flex.sourceforge.net/>, The Flex Project, 2008