siC: Uma linguagem baseada em C incluindo pilha e fila como tipos primitivos

Gabriella de Oliveira Esteves, 110118995

[1]Departamento de Ciência da Computação - Universidade de Brasília

1. Objetivo

Este trabalho visa projetar e construir uma nova linguagem chamada de siC - Structures in C, baseada na linguagem C. O siC acrescenta três estruturas de dados, pilha e fila, como tipos de dados primitivos e, para manipulá-las, adiciona certas operações próprias para tal.

2. Introdução

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

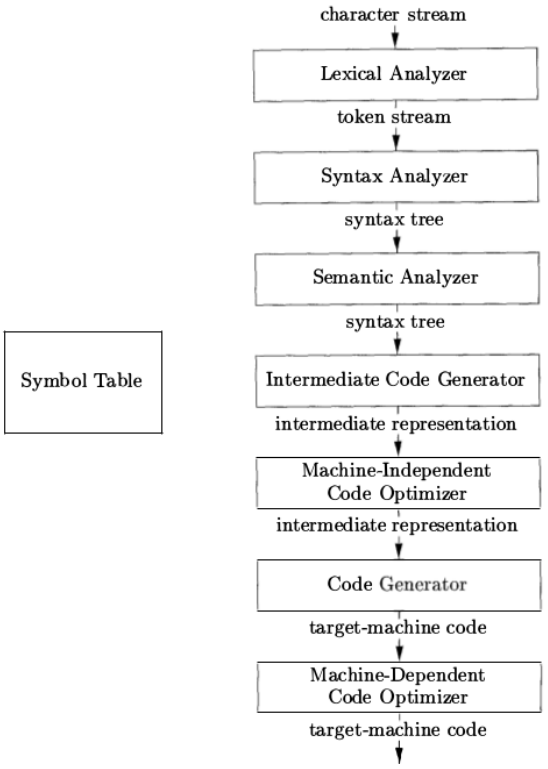

Figura 1. Fases de um compilador

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 gramátical dos tokens em forma de árvore;

3 **Analisador 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, bem como uma breve descrição de sua semântica. Como pilha e fila são duas das estruturas de dados mais básicas, é possível dizer que siC se destina a inúmeras áreas de Ciência da Computação, como sistemas operacionais (onde a fila é usada para organizar prioridades dos processos, por exemplo), compiladores (onde a pilha é usada para ordenar a execução dos métodos/funções), etc.

Dois grandes motivos sustentam a escolha do tema deste projeto. Primeiro, uma vez que as estruturas de dados pilha e fila fazem 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 das estruturas de dados báscias, e, de maneira geral, pode ser mais *user-friendly*. Nesse sentido, o usuário (da linguagem) leigo deverá entender como cada estrutura funciona, bem como suas vantagens/desvantagens e usabilidade; porém a implementação de cada uma estará a cargo da própria siC.

3. 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 linguagen 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 topo da pilha ou início da fila caso estes sejam os tipos do *identifier*. Segue abaixo a gramática proposta, em que a variável inicial é *function*.

```
token: WHILE, IF, ELSE, RETURN, STACK, TOP, QUEUE, FIRST, VOID, BOOL, INT, CHAR

function
   → type identifier '(' argument ')' '{' statement 'RETURN' identifier ';' '}'
    | type identifier '(' ')' '{' statement '}'

identifier
   → letra(letra | digito)*
```

```
letra
   → 'a' | 'b' | ... | 'z'

digito
   → '0' | '1' | ... | '9'

identifier_struct
   → identifier
    | identifier '.' TOP
    | identifier '.' FIRST

type_struct
   → type_simple
    | type_stack
    | type_queue

type_simple
   → VOID
    | BOOL
    | INT
    | CHAR
```

Existem dois novos tipos de dados, *STACK* e *QUEUE*, que serão compostas por tipos simples de dados apenas (ou seja, não será possível criar uma variável do tipo pilha onde seus elementos também são pilhas). Caso a variável seja do tipo pilha, ela poderá obter o topo através do comando "identifier.TOP". Caso seja fila, poderá obter o primeiro elemento com "identifier.FIRST".

```
type_stack
   → STACK '<' type_simple '>'

type_queue
   → QUEUE '<' type_simple '>'


argument
   → type_struct identifier
    | type_struct identifier ',' 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.

```
statement
   → declare_identifier
    | while_expression
    | if_expression
    | math_expression
    | identifier_struct_expression
    | ε
```

```
declare_identifier
   → argument ';'

while_expression
   → WHILE '(' compare_expression ')' '{' statement '}'

if_expression
   → IF '(' compare_expression ')' '{' statement '}'
    | IF '(' compare_expression ')' '{' statement '}' ELSE '{' statement '}'

compare_expression
   → identifier_struct compare_assignment identifier_struct

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

math_expression
   → identifier '=' factor ';'

factor
   → identifier_struct
    | '(' number_expression ')'

number_expressions
   → number_expression '+' term
    | number_expression '-' term
    | term

term
   → term '*' factor
    | term '/' factor
    | factor


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

4. 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 em um identificador do tipo struct B, a atribuição deve ser do tipo B = B + A, onde A deverá ter tipo compatível com o de B;
- Para remover o topo/início de um identificador do tipo struct B, a atribuição deve ser do tipo B = B - A, onde A deverá ter tipo compatível com o de B;
- Nenhuma operação matemática (*math_expression*) pode conter um identificador do tipo pilha ou fila, apenas o topo ou início dos mesmos.

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

```
1  VOID main () {
2      STACK<INT> s;
3      INT sum, INT lixo;
4
5      s = s + 0;
6      s = s + 1;
7      s = s + 2;
8      s = s + 3;
9      sum = 0;
10
11     WHILE (s.TOP != 0) {
12         sum = (sum + s.TOP);
13         s = s - 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.