

**UNIDADE CURRICULAR:** Compilação

**CÓDIGO:** 21018

**DOCENTE:** Professor Carlos Morais

**NOMES** e **N.º DE ESTUDANTE:** (Alt + C ++ Elite)

- Gonçalo Caraça, 2000130

- Inês Oliveira, 2001090

- Joana Martins, 2003351

- João Carvalho, 2103537

- Mário Carvalho, 2000563

**CURSO:** Licenciatura em Engenharia Informática

**DATA DE ENTREGA:** 14 de junho de 2023

**TRABALHO / RESOLUÇÃO:**

Tendo em conta que o relatório final trata o conjunto do trabalho realizado ao longo do semestre, procedeu-se a uma explicação mais detalhada do processo relativo às etapas de construção de um compilador para linguagem YAIL.  
Assim, após análise e avaliação das possibilidades de linguagem e preferências dos elementos do grupo, escolheu-se recorrer à linguagem C e às ferramentas ***Flex*** e ***Bison***, pelo potencial de automatização das tarefas necessárias que apresentavam para facilitar a implementação de código executável final.

Durante a construção da Análise Léxica recorreu-se à construção de caracteres literais e de palavras-chave imprescindíveis para a identificação de “*tokens*”, após leitura do texto de entrada. Em parte, recorreu-se ao uso de regras estipuladas em “*Regular Expressions*”, que permitissem produzir sequencias de palavras (a partir de símbolos, números, …) subsequentemente identificadas na Análise Sintática. Como exemplo, tem-se por objetivo que o caractere “#” indique constantemente que seja o início de um comentário. Obtendo este ficheiro final, após execução do comando associado à ferramenta Flex para obter um ficheiro final “.lex”.  
De seguida procedeu-se à construção de regras gramaticais que obedecem às critérios de escrita do YAIL. Recorrendo à leitura da sequência dos “*tokens”* gerados anteriormente pela análise léxica, verifica-se assim, se a escrita de código segue a gramática da linguagem pedida, recorrendo à ferramenta Bison, para obter o ficheiro executável (“.y”), e ou identificação de erros.

Numa primeira etapa (para fase de trabalho do E-Fólio A), os ficheiros finais obtidos apresentavam-se como promissores, até se começar a implementar a parte responsável pela Análise Semântica. Que, após alterações das condições de YAIL, se procedeu a nova reestruturação de ambos os ficheiros desenvolvidos até esta fase.

Análise Semântica:

* + A análise semântica é responsável por verificar se o programa segue as regras semânticas da linguagem.
  + Ela identifica erros como uso incorreto de tipos, escopo de variáveis, verificação de compatibilidade, entre outros.
  + Nessa etapa, é criada uma árvore sintática abstrata (AST) que representa a estrutura do programa.

1. Geração de Código Intermediário:
   * O código intermediário é uma representação de nível mais baixo do programa, independente da arquitetura do processador.
   * Pode ser uma representação em forma de árvore, notação polonesa reversa (RPN) ou código de três endereços.
   * O código intermediário facilita a otimização e a geração de código final.
2. Otimização de Código:
   * A otimização de código visa melhorar o desempenho e a eficiência do programa.
   * Técnicas como análise de fluxo de dados, eliminação de código morto, propagação de constantes, entre outras, podem ser aplicadas nessa etapa.
3. Geração de Código:
   * Nessa etapa, o código final é gerado a partir do código intermediário otimizado.
   * O código é específico para a arquitetura do processador de destino.
   * As técnicas de geração de código podem variar dependendo da complexidade do compilador e do alvo.
4. Vinculação e Execução:
   * Após a geração do código, é necessário vincular o código-fonte com bibliotecas e outros módulos externos.
   * Em seguida, o programa é executado na máquina de destino.

melhorias

Evolução para Global

Resultado Final

Apreciação do executável e como o compilar (comandos GCC, ..) => compileCodigo.sh)

MELHORIAS CORRIGIDAS  
MELHORIAS FUTURAS:

1) Excluir do regex IDENT a possibilidade de escrever INT; Float ou Bool

=> identificador [\_a-zA-Z\\_]+([0-9]?|[\_a-zA-Z\\_]?) => JOao

8) Construir zona corrigida para validar tamanhos de vetores => Todos?

11) IMPLEMENTAR TAC, E TRANSFORMAÇÃO ASSEMBLY com funções em C para cada parte => TODOS?

10) CORRIGIR ERROS E GCC ?!?!?! COMO FAZER ?

2) Tratamento de erros e mensagens certas, com correções de ambiguídades

via comando => "win\_bison --report=all --warninfgs=all --debug -o %(Filename.cpp)"

/\*/\*/\*/\*/\*/\*/\*/\*/\*/\*/\*/\*/\*/\*/\*/\*/\*/\*/\*/\*/\*/\*/\*/\*/\*/\*/\*/\*/\*/\*/\*/\*/\*/\*/\*/\*/\*/

DONE 4) Assimilar diferenças ou associar:

GLOBAL ABRECHAVETA declara\_variavel FECHACHAVETA fim\_linha

CONST ABRECHAVETA declaracao\_atribuicao FECHACHAVETA fim\_linha

CONCLUSÃO : ESTÃO DIFERENTES Não me parece que permita associar!

DONE 3) Rever problema de números negativos não serem interpretados como subtração:

No lex está previsto os valores decimais e inteiros terem ou não um - atrás.

DONE 5) Corretivo quanto aos comparativos algébricos para fazer funcionar na ordem correta :

"+="|"-="|"\*="|"/="|"%=" { return (OPERADORLOGICO); } OK

CORRIGIR

"+"|"-"|"\*"|"/"|"%" { return (OPERADOR); }

OPERADOR ISOLADO

"++"" { return (INCREMENTO); }

"--"" { return (DECREMENTO); }

"+" { return (OPERADOR); }

"-" { return (MENOS); }

"\*" { return (MULTIPLICA); }

"/" { return (DIVIDE); }

"%" { return (MODULO); }

%type<tipoint>variavel\_int /\* TODO: A VERIFICAR SE FAZ SENTIDO!!!!\*/

%type<tipostring>variavel\_string /\* TODO: A VERIFICAR SE FAZ SENTIDO!!!!\*/

%type<tipofloat>variavel\_float /\* TODO: A VERIFICAR SE FAZ SENTIDO!!! \*/

DONE 6) SUBSTITUIR? APAGAR? => substituido por OPERADOR OPERADOR

"++" { return (INCREMENTO); }

"--" { return (DECREMENTO); }

DONE 7) Ordenar funções, e simplificar nomes se possivel

DONE 9) SEEGUNDO NOVO ENUNCIADO o ponto "." tem de sair, certo?

valores:

ABREPARENT valores\_dentro FECHAPARENT // (INTEIRO, FLOAT, BOOLEANO)

| ABREPARENT valores\_dentro FECHAPARENT // (q) (q.x) (q.p.x)

| valores\_dentro

;

valores\_dentro:

valor // valor = INTEIRO, FLOAT, BOOLEANO

| IDENT // q

| IDENT PF IDENT // q.x

| IDENT PF IDENT PF IDENT // q.p.x

**BIBLIOGRAFIA**

Aho, A., S. Lam, M., Ravi , S., & D. Ullman, J. (2008). *Compilers: principles, techniques and tools* (2ª ed.). (A. Wesley, Ed., & D. Vieira, Trad.) Universidade Federal de Minas Gerais, Brasil: Pearson.

Reis Santos, P., & Thibault, L. (2015). *Compiladores.* FCA.