

Compiladores – 2019/2

Trabalho Prático T2

24 de setembro de 2019

1 Objetivo

O objetivo deste trabalho é a criação de um analisador sintático (*parser*) para a linguagem C-Minus, como segundo componente do *front-end* de um compilador.

2 Convenções Sintáticas de C-Minus

A sintaxe da linguagem está apresentada em notação BNF abaixo, aonde os terminais (*tokens*) estão escritos em fonte **mono-espçada** e os não-terminais em *itálico*.

```
program → func-decl-list
func-decl-list → func-decl-list func-decl | func-decl
func-decl → func-header func-body
func-header → ret-type ID ( params )
func-body → { opt-var-decl opt-stmt-list }
opt-var-decl → ε | var-decl-list
opt-stmt-list → ε | stmt-list
ret-type → int | void
params → void | param-list
param-list → param-list , param | param
param → int ID | int ID [ ]
var-decl-list → var-decl-list var-decl | var-decl
var-decl → int ID ; | int ID [ NUM ] ;
stmt-list → stmt-list stmt | stmt
stmt → assign-stmt | if-stmt | while-stmt | return-stmt | func-call ;
assign-stmt → lval = arith-expr ;
lval → ID | ID [ NUM ] | ID [ ID ]
if-stmt → if ( bool-expr ) block | if ( bool-expr ) block else block
block → { opt-stmt-list }
while-stmt → while ( bool-expr ) block
return-stmt → return ; | return arith-expr ;
func-call → output-call | write-call | user-func-call
input-call → input ( )
output-call → output ( arith-expr )
write-call → write ( STRING )
user-func-call → ID ( opt-arg-list )
opt-arg-list → ε | arg-list
arg-list → arg-list , arith-expr | arith-expr
bool-expr → arith-expr < arith-expr | arith-expr <= arith-expr | arith-expr > arith-expr
               | arith-expr >= arith-expr | arith-expr == arith-expr | arith-expr != arith-expr
arith-expr → arith-expr + arith-expr | arith-expr - arith-expr
               | arith-expr * arith-expr | arith-expr / arith-expr
               | ( arith-expr ) | lval | input-call | user-func-call | NUM
```

A gramática acima possui ambiguidades que levam a conflitos de *shift-reduce*. Utilize os comandos do **bison** para definição de prioridades de operadores (`%left`, etc) para remover as ambiguidades da gramática. **Não envie um trabalho para avaliação com conflitos de *shift-reduce* ou *reduce-reduce*.**

3 Implementando e Testando o Trabalho

As convenções léxicas da linguagem C-Minus já foram apresentadas na especificação do Trabalho 1. Você deve adaptar o *scanner* desenvolvido anteriormente para compor o *parser* deste trabalho.

Utilize as demais opções do **bison** como demonstrado pelo professor nos exemplos das aulas de laboratório.

O seu programa de entrada é lido da entrada padrão (*stdin*), como abaixo:

```
$ ./trab2 < program.cm
```

Se o programa estiver correto, o seu *parser* deve exibir uma mensagem indicando que o programa foi aceito, como abaixo:

```
$ ./trab2 < program.cm
PARSE SUCCESSFUL!
```

Se o programa possuir erros léxicos, exiba uma mensagem informativa como abaixo:

```
$ ./trab2 < program.cm
SCANNING ERROR (XX): Unknown symbol SS
```

Aonde XX é a linha aonde o símbolo desconhecido SS apareceu.

Se o programa possuir erros sintáticos, exiba uma mensagem informativa como abaixo:

```
$ ./trab2 < program.cm
PARSE ERROR (XX): syntax error, unexpected UT, expecting ET
```

Aonde UT e ET são os tipos de *tokens* lido e esperado, respectivamente. Utilizando as opções `%define parse.error verbose` e `%define parse.lac full`, a mensagem depois do `:` já é gerada automaticamente pelo **bison**. Neste caso, basta definir a função de erro como abaixo:

```
// Error handling.
void yyerror (char const *s) {
    printf("PARSE ERROR (%d): %s\n", yylineno, s);
}
```

Observações importantes:

- O seu *parser* pode terminar a execução ao encontrar o primeiro erro no programa de entrada.
- Os arquivos de entrada de exemplo são os mesmos do Trabalho 1.
- Valide o seu programa usando os gabaritos disponibilizados na sala da disciplina no AVA. Garanta que seu programa gerará exatamente o mesmo resultado do gabarito usando o utilitário **diff** (no Linux). Exemplo:

```
./trab2 < in/c01.cm | diff out2/c01.out -
```

ATENÇÃO: seu programa deve produzir como saída (NA TELA) SOMENTE o resultado no formato acima. Nada além disto. Ou seja, nenhuma mensagem e nenhuma formatação adicional deverá ser exibida. Isto é absolutamente necessário porque será usada uma bateria de testes para validação de seu trabalho, que verifica se sua resposta está correta baseado na saída do seu programa.

- Ao submeter o seu trabalho para correção, além dos códigos-fonte envie um arquivo **Makefile** que gera como executável para o seu *parser* um arquivo de nome **trab2**.

4 Regras para Desenvolvimento e Entrega do Trabalho

- **Data da Entrega:** O trabalho deve ser entregue até às 23:55 h do dia 07/10/2019 (segunda-feira). Não serão aceitos trabalhos após essa data.
- **Grupo:** O trabalho é **individual**.
- **Linguagem de Programação e Ferramentas:** Para implementar o seu analisador sintático você deve obrigatoriamente usar o **bison**.
- **Como entregar:** Pela atividade criada no AVA. Envie um arquivo compactado com todo o seu trabalho.
- **Recomendações:** Modularize o seu código adequadamente. Crie códigos claros e organizados. Utilize um estilo de programação consistente. Comente o seu código extensivamente. Não deixe para começar o trabalho na última hora.

5 Avaliação

- O trabalho vale 1.5 ponto na média parcial do semestre.
- Trabalhos com erros de compilação receberão nota zero. Isso inclui trabalhos com conflitos de *shift-reduce* ou *reduce-reduce*.
- Caso seja detectado plágio (entre alunos ou da internet), todos os envolvidos receberão nota zero.
- Serão levadas em conta, além da correção da saída do seu programa, a clareza e simplicidade de seu código.
- A critério do professor, poderão ser realizadas entrevistas com os alunos, sobre o conteúdo do trabalho entregue. Caso algum aluno seja convocado para uma entrevista, a nota do trabalho será dependente do desempenho na entrevista. (Vide item sobre plágio, acima.)