

Emanuel Felipe Giroldo Mazzer

Compilador Linguagem T++

Relatório técnico da Atividade Prática Supervisionada solicitado pelo professor Rogério Aparecido Gonçalves na disciplina de Compiladores do curso Bacharelado em Ciência da Computação da Universidade Tecnológica Federal do Paraná.

Universidade Tecnológica Federal do Paraná – UTFPR

Departamento Acadêmico de Computação – DACOM

Bacharelado em Ciência da Computação – BCC

Campo Mourão

Julho / 2017

Resumo

Neste trabalho foi realizado a implementação de um compilador para a linguagem de programação T++. A implementação foi dividida em quatro partes, análise léxica, que tem como objetivo ler o código fonte e identificar os tokens do programa, análise sintática, que é responsável por verificar se os tokens contidos no programa fonte formam um programa válido, a análise semântica, que analisa as estruturas construídas pelo analisador sintático, e por final a geração de código, aonde ocorre a transformação da árvore sintática em uma representação intermediária do código fonte.

Palavras-chave: análise léxica. análise sintática. análise semântica. geração de código.

Lista de abreviaturas e siglas

PLY	Python Lex Yacc
-----	-----------------

Sumário

1	Introdução	5
2	Objetivos	5
3	Fundamentação	5
3.1	Linguagem t++	5
3.2	Análise Léxica	6
3.3	Análise Sintática	7
3.4	Análise semântica	8
3.5	Geração de código intermediário	10
4	Materiais	11
5	Discussão dos Resultados	11
	Referências	11

1 Introdução

Um compilador é um programa de computador que traduz um programa escrito em uma linguagem de alto nível para um programa equivalente em código de máquina. Geralmente o compilador não produz diretamente o código de máquina, mas sim um programa em linguagem assembly, que semanticamente é equivalente ao programa em linguagem de alto nível. O programa em assembly então é traduzido para o programa em linguagem de máquina. Geralmente um compilador possui etapas que devem ser executadas até que o código final seja obtido, essas etapas são chamadas de fases de compilação e podem ser divididas basicamente em, análise léxica, análise sintática, análise semântica e geração de código.

Neste relatório será explicado detalhadamente a implementação de todas essas etapas, e também será explicada a linguagem T++, que é uma linguagem de programação criada pelo professor Rodrigo Hubner, para a realização de testes no compilador criado.

2 Objetivos

O objetivo deste trabalho consiste na criação de um compilador para a linguagem procedural T++, passando pela implementação de todas as suas etapas e obtendo finalmente um código em linguagem simbólica.

3 Fundamentação

Como já dito, um compilador possui várias etapas até que o código final seja gerado, a seguir será explicado detalhadamente todas as etapas de compilação.

3.1 Linguagem t++

A linguagem T++ é uma linguagem procedural capaz de definir funções, condições, operações matemáticas, etc. Como várias outras linguagens T++ possui várias palavras-chaves e símbolos reservados pela linguagem. As palavras e símbolos reservados estão representados na tabela a seguir.

Tabela 1: Símbolos da linguagem

Símbolos	Significado
+	Soma
-	Subtração
*	Multiplicação
/	Divisão
=	Igualdade
,	Virgula
:=	Atribuição
<	Menor
>	Maior
<=	Menor-igual
=>	Maior-igual
(Abre-parênteses
)	Fecha-Parênteses
:	Dois Pontos
[Abre-Colchetes
]	Fecha-Colchetes

Tabela 2: Palavras da linguagem

Palavra	Representação
Se	IF
então	THEN
senão	ELSE
fim	END
repita	REPEAT
flutuante	FLOAT
retorna	RETURN
até	UNTIL
leia	READ
escreve	WRITE
inteiro	INT

3.2 Análise Léxica

A análise léxica é o processo de analisar a entrada de caracteres, como um código fonte de um computador, e produzir uma sequência de símbolos chamados token, esses tokens podem ser manipulados mais facilmente por um parser. O componente responsável por realizar essa função é conhecido como analisador léxico.

O analisador léxico faz uma varredura do programa fonte, caractere por caractere, e traduz em uma sequência de tokens. É nessa fase que são reconhecidas as palavras reservadas, constantes, identificadores, e outras palavras que pertencem a linguagem de programação. Além disso, o analisador léxico é responsável por tratar os espaços, remover

os comentarios, e realizar a contagem do numero de linhas que o programa possui. É na análise léxica que é verificado se algum caracteres não faz parte do alfabeto determinado pelo programados. Para uma melhor visualização foram motados automatos finitos determinísticos de todas as expressões regulares utilizadas na linguagem.

3.3 Análise Sintática

O analisador sintatico também conhecido como *parser* tem como tarefa principal determinar se o programa de entrada, representado pelo fluxo de tokens, possui as sentenças válidas para a linguagem de programação. No caso de analisadores sintáticos *top – down*, temos a opção de escreve-los à mão ou gera-los de forma automática, mas os analisadores *bottom – up* só podem ser gerados automaticamente. A maioria dos métodos de análise sintática cai em uma dessas duas classes. Existem disponíveis uma série de geradores automaticos de analisadores sintaticos, como por exemplo, o YACC, BISON e JavaCC. Neste trabalho foi utilizado o python YACC.

Na figura 1 é mostrado um exemplo da saída sintatica para o seguinte código:

{exemplo tpp}

```
inteiro principal()
  inteiro: x
  inteiro: y
  inteiro: z
  x:=10
  y:=10
  z := x+y
  retorna(z)
fim
```

[illegible]

Figura 1: Exemplo de saída da análise sintática.

3.4 Análise semântica

Análise semântica é a terceira fase da compilação, é nela, onde logicamente, são verificados os erros semânticos, como por exemplo, a divisão de um numero inteiro por um numero float. A análise semântica trata a entrada sintática e a transforma em uma representação mais simples e mais adaptada a geração de código. Além disso, esta camada do compilador fica encarregada de analisar a utilização dos identificadores e de ligar cada delas a sua declaração. É nessa etapa de compilação que é verificado se cada expressão definida tenha o tipo esperado conforme as regras da linguagem.

Para armazenar toda essa informação o analisador semântico possui uma tabela, sendo assim o analisador consegue verificar se uma variável já foi declarada ou não, e é também a partir dessa tabela que é verificado se uma variável está sendo utilizada no código ou não. Nesta implementação a tabela possui os seguintes campos: classe(variável ou função), nome da variável, se é utilizada, se é atribuída, tipo da variável. A figura(2) mostra a tabela de variáveis. utilizada.

```
{'global-a': ['variavel', 'a', True, True, 'flutuante', 0]}
```

Figura 2: Exemplo de saída da análise semântica

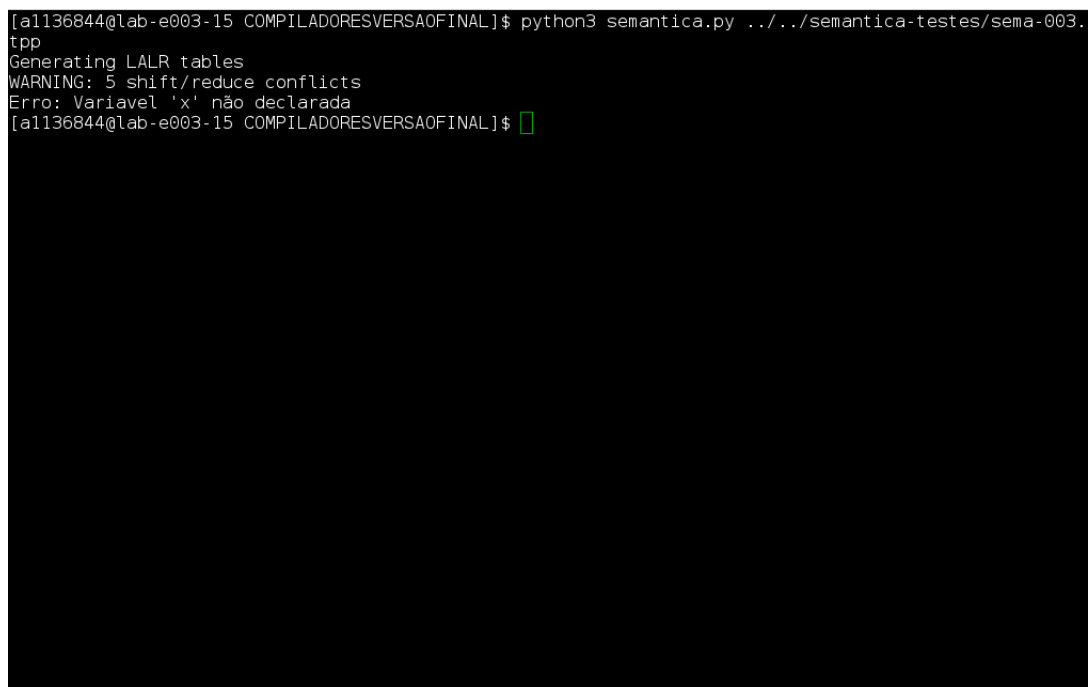
A figura(4) mostra a saída da análise semântica para o seguinte código:

```
{exemplo tpp}
```



```
inteiro principal()  
    inteiro: y  
    inteiro: z  
    x:=10  
    y:=10  
    z := x+y  
    retorna(z)  
fim
```

A análise semântica deve retornar um erro, pois a variável `x` é utilizada sem ser declarada.



```
[a1136844@lab-e003-15 COMPILADORESVERSAOFINAL]$ python3 semantica.py ../../semantica-testes/sema-003.tpp  
Generating LALR tables  
WARNING: 5 shift/reduce conflicts  
Erro: Variavel 'x' não declarada  
[a1136844@lab-e003-15 COMPILADORESVERSAOFINAL]$
```

Figura 3: Exemplo de saída da análise semântica

Quando nenhum erro é encontrado a análise semântica deve retornar apenas a tabela de símbolos gerada para aquele código. Segue descrito quais *warnings* e erros foram implementados para serem captados nesta análise:

- Um *warning* é mostrado quando uma variável for declarada mais de uma vez
- Um *warning* é mostrado quando uma função é criada e nunca utilizada
- Um *warning* é mostrado quando uma variável é declarada mas nunca utilizada
- Um *warning* é mostrado quando ocorrer uma coerção implícita de tipos (inteiro<->flutuante)
- Um *warning* é mostrado quando há chamada recursiva para a função principal

- Um *warning* é mostrado quando tipos diferentes são passados por parâmetros em uma chamada de função
- um erro é mostrado quando a função principal não for declarada
- Um erro é mostrado quando uma variável é utilizada mas nunca declarada
- Um erro é mostrado quando uma variável é declarada com o mesmo nome de uma função
- Um erro é mostrado quando uma função já foi declarada
- Um erro é mostrado quando uma variável não foi inicializada
- Um erro é mostrado quando uma função é utilizada mas nunca declarada
- Um erro é mostrado quando o número de parâmetros passados é diferente do número de parâmetros esperados
- Um erro é mostrado quando o valor de um índice não for inteiro

3.5 Geração de código intermediário

Nessa etapa, ocorre a transformação da árvore sintática em uma representação intermediária do código fonte. Esta linguagem intermediária é mais próxima da linguagem objeto que o código fonte, mas ainda permite uma manipulação mais fácil do que o código assembly ou código de máquina. A geração de código é uma das fases mais complexas de um compilador, pois não depende apenas das características da linguagem fonte, mas também de informações e detalhes da arquitetura alvo, da estrutura do ambiente de execução e do Sistema Operacional da máquina alvo. Um compilador tipicamente quebra essa fase em vários passos que usam diversas estruturas de dados intermediárias, e frequentemente requerem alguma forma de código abstrato, denominado código intermediário.

Um tipo popular de código intermediário é conhecido como *código de três endereços*. Neste tipo de código uma sentença tem a forma $X := A \text{ op } B$, onde X , A e B são operandos e op uma operação qualquer. Uma maneira de representar sentenças de três endereços é pelo uso de quádruplas (operador, argumento1, argumento2, resultado). Este esquema de representação é preferido por diversos compiladores, pois o código intermediário pode ser rearranjado de maneira fácil e conveniente, permitindo uma facilidade na otimização de código.

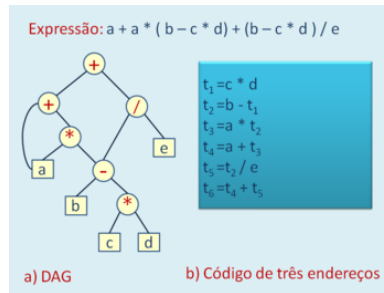


Figura 4: Exemplo de código de três endereços e o seu DAG

4 Materiais

Para a realização da atividade foram utilizados os seguintes materiais:

- Python Lex Yacc - PLY
- Python3
- LLVM - Low Level Virtual Machine

5 Discussão dos Resultados

A implementação completa de um compilador foi um grande desafio, repleto de dificuldades e aprendizados, sempre que a implementação de uma nova etapa (léxica, sintática, semântica, geração de código) era iniciada, muitas dúvidas e contratempos apareciam, porém, depois que um tempo era dedicado na resolução dos problemas, tudo começava a fazer sentido e a implementação se mostrava mais simples do que de fato parecia. A análise semântica foi a etapa que houve maior dedicação de tempo para sua resolução, os resultados foram ótimos e superiores ao esperado. A geração de código não foi implementada por inteiro, devido a sua grande complexidade e a falta de tempo para a entrega final, as coisas não saíram como o esperado.

A elaboração de um compilador não é algo trivial, ao mesmo tempo que se mostra uma tarefa difícil, também se mostra divertido e que demanda uma grande dedicação de tempo, principalmente na fase da geração de código intermediário, aonde uma grande quantidade de funções e módulos da llvm tinham que ser aprendidos.