



Centro Federal de Educação Tecnológica de Minas Gerais

Departamento de Computação

Curso de Engenharia de Computação

Josué Rocha Lima
Felipe Freitas

Trabalho Prático: implementação de um analisador semântico

KPiler: compilador para a linguagem K

Relatório do trabalho prático apresentado à
disciplina de compiladores.

Orientador: Kécia Aline Marques Ferreira

Belo Horizonte

2017

Sumário

1	INTRODUÇÃO	3
1.1	Proposta de trabalho prático	4
2	METODOLOGIA	7
2.1	Projeto do analisador semântico	7
2.2	Implementação do analisador semântico	11
2.2.1	Arquitetura da implementação	11
2.2.2	Recursos adicionais	12
3	INSTRUÇÕES DE UTILIZAÇÃO	13
4	AVALIAÇÃO DOS RESULTADOS	14
4.1	Teste 1	14
4.2	Teste 2	16
4.3	Teste 3	17
4.4	Teste 4	19
4.5	Teste 5	21
4.6	Teste 6	22
5	CONCLUSÃO	24
	REFERÊNCIAS	25

1 Introdução

Com o intuito de reduzir a complexidade da implementação de sistemas de software, foi proposta a utilização de linguagens próximas da linguagem humana, denominadas linguagens de alto nível. Entretanto, para possibilitar a implementação de software dessa maneira, é necessário a tradução para uma linguagem que possa ser executada diretamente no *hardware* (AHO; SETHI; ULLMAN, 2007).

O compilador é um sistema de *software* responsável pela tradução automática de um programa em linguagem fonte de alto nível, para um programa semanticamente equivalente em uma linguagem alvo, conforme pode-se observar na Figura 1. Deste modo, os compiladores são considerados *software* básico para a computação.

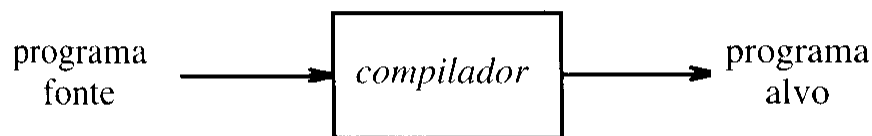


Figura 1 – Tradução do programa fonte para a linguagem alvo realizada pelo compilador (AHO; SETHI; ULLMAN, 2007).

O compilador é comumente dividido em módulos para reduzir a complexidade do seu processo de implementação. Essa divisão é frequentemente compreendida pelos seguintes módulos:

- **Front-end (Análise)**
 - **Analisador léxico:** lê o programa fonte caractere a caractere e agrupa o programa em sequências significativas denominadas lexemas. Para cada um dos lexemas é gerado um *token* no formato <nome , valor>. O analisador léxico também é responsável por reconhecer e ignorar comentários e gerenciar numeração de linhas.
 - **Analisador sintático:** recebe o fluxo de *tokens* gerado pelo analisador léxico e verifica se a sequência está de acordo com a estrutura sintática da linguagem. É o núcleo do compilador.
 - **Analisador semântico:** Utiliza a tabela de símbolos e árvore sintática para verificar se o programa-fonte atende às regras semânticas da linguagem, como regras de compatibilidade de tipos.

- **Gerador de código intermediário:** Código em representação intermediária em baixo nível é gerado para auxiliar a geração de código e otimização.
- **Back-end (Síntese)**
 - **Gerador de código:** mapeia a linguagem intermediária para a linguagem objeto.

A Figura 1 ilustra o fluxo entre os módulos do compilador:

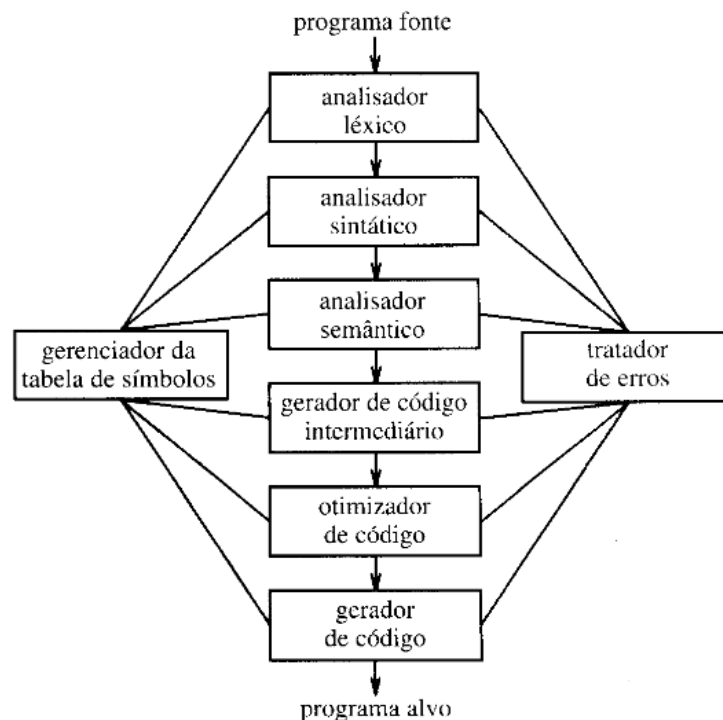


Figura 2 – Divisão em módulos do compilador (AHO; SETHI; ULLMAN, 2007).

1.1 Proposta de trabalho prático

Foi proposto o projeto e implementação de um compilador para uma linguagem hipotética K como trabalho prático e didático para a disciplina de Compiladores. Devido à complexidade do problema de compilação, o projeto foi dividido em quatro componentes com suas respectivas entregas:

1. Analisador léxico e tabela de símbolos;
2. Analisador sintático;
3. Analisador semântico;

4. Gerador de código.

O trabalho entregue anexo à este relatório corresponde ao analisador semântico (3). A análise semântica é um processo que consiste em verificar a consistência semântica do programa-fonte com a definição da linguagem-fonte utilizando a árvore de sintaxe gerada pelo analisador sintático e a tabela de símbolos (AHO; SETHI; ULLMAN, 2007). Durante esse processo, o analisador semântico também coleta informações de tipo e as adiciona à tabela de símbolos para auxiliar durante o processo de geração de código.

Existem dois tipos de análise semântica: dinâmica e estática. A análise semântica dinâmica é executada em tempo de execução, como PROLOG e PHP. Por outro lado, a análise semântica estática é executada em tempo de compilação, como nas linguagens C e Pascal.

A verificação de tipos é uma das responsabilidades principais do analisador semântico, na qual verifica-se se cada operador possui operandos consistentes com a especificação (AHO; SETHI; ULLMAN, 2007). A linguagem Java, por exemplo, requer que uma condição em dado comando *if* possua tipo resultante lógico (*booleano*). Deste modo, deve-se exibir um erro caso o usuário utilize uma expressão com tipo *string* em um comando *if*. Para isso, o analisador semântico verifica o tipo das expressões por meio de produções da gramática e o tipo dos identificadores por meio da tabela de símbolos. O verificador de tipos também deve acrescentar o tipo dos identificadores às respectivas entradas na tabela de símbolos.

Deste modo, para realizar a checagem de tipos, é necessário ao compilador assinalar um tipo à cada componente do programa-fonte, iniciando a partir das produções de palavras-chave que expressam tipos. É possível obter o tipo de cada expressão por síntese ou inferência. A síntese consiste em obter o tipo de determinada expressão a partir de suas subexpressões. Por exemplo, em uma determinada especificação de semântica, se uma expressão consiste na soma de vários identificadores de tipo inteiro, à essa expressão também é assinalado o tipo inteiro (AHO; SETHI; ULLMAN, 2007).

Já a inferência de tipos consiste em determinar o tipo de determinada construção de acordo com a maneira com que é utilizada. Linguagens como ML, que fazem checagem de tipos mas não requerem a declaração explícita de nomes de tipos utilizam inferência (AHO; SETHI; ULLMAN, 2007).

A verificação de tipos é frequentemente implementada compiladores com análise sintática recursiva descendente por meio de esquemas de tradução baseados em gramáticas de atributos. Os esquemas de tradução utilizados em análise semântica consistem na adição de regras semânticas à cada produção da gramática. Estes esquemas de tradução sintetizam tipos para cada produção e verificação se esses tipos atendem às especificações da gramática (AHO; SETHI; ULLMAN, 2007).

Cumprir observar que durante o processo de análise semântica ocorrem a análise

contextual, e as verificações de unicidade e classe. A verificação de unicidade consiste em assegurar que cada identificador foi definido somente uma vez. Durante a execução de produções de declaração de identificadores, consulta-se a tabela de símbolos para constatar se o identificador já foi definido previamente. Caso ele não tenha sido definido, seu tipo é adicionado à tabela de símbolos, caso contrário, uma mensagem de erro é exibida.

Ademais, a análise contextual visa assegurar que todos os identificadores utilizados foram declarados previamente. Isso pode ser alcançado por meio de consultas à tabela de símbolos.

Por fim, a verificação de classe consiste em identificar se determinado identificador representa uma variável, procedimento ou função. Isso pode ser alcançado realizando a atribuição de classes de acordo com o nó da árvore de sintaxe ou a função correspondente à produção do analisador sintático à qual o identificador foi assinalado.

2 Metodologia

Para verificação da conformidade do programa-fonte com as regras semânticas da linguagem K , foram incorporados fragmentos de código correspondentes às ações semânticas à cada função do analisador sintático recursivo descendente desenvolvido na etapa anterior deste trabalho. Foi construído um projeto para o analisador semântico para servir como referencial para o desenvolvimento e para evitar erros nos momentos finais do desenvolvimento.

2.1 Projeto do analisador semântico

O projeto do analisador semântico foi iniciado pelo estudo de cada produção da gramática da linguagem K , o qual visou identificar produções nas classes de: comandos, expressões e tipos. Posteriormente, cada operador dentro do conjunto de expressões e seus respectivos tipos de operandos válidos foram analisados.

Percebeu-se que a declaração de identificadores ocorria na produção *decl*. Durante o processamento dessa produção, o tipo especificado em *decl* é assinalado à todos os identificadores produzidos na produção *identifier-list* por meio de atributos herdados passados para as produções seguintes de identificadores.

Para implementar verificação de unicidade, durante o processamento das produções *identifier-list* e *possible-ident*, as informações do identificador declarado são consultadas na tabela de símbolos, com o intuito de detectar uma possível redeclaração.

Para realizar análise contextual, foi incorporado às produções que caracterizam uma referência à um *token* identificador (*factor* e *read*) uma consulta à tabela de símbolos que visa detectar uma referência à uma variável não declarada.

Ao restante das produções foram adicionados atributos sintetizados para possibilitar a análise de tipos. A gramática com as respectivas ações semânticas de cada produção pode ser observada abaixo. Algumas ações semânticas são exibidas após a gramática para facilitar a visualização.

$$\langle program_prime \rangle ::= \langle Program \rangle \text{ '$' } \quad (1)$$

$$\langle Program \rangle ::= \text{'program'} \langle decllist \rangle \langle stmt-list \rangle \text{'end'} \quad (2)$$

$$\langle decllist \rangle ::= \langle decl-list \rangle \quad (3)$$

$$| \text{'\lambda'} \quad (4)$$

$$\langle decl-list \rangle ::= \langle decl \rangle \langle decllist \rangle \quad (5)$$

$$\langle decl \rangle ::= \langle type \rangle \langle identifier-list \rangle \{ identifier-list.tipo = type.tipo \} \quad (6)$$

$$\begin{aligned} \langle identifier-list \rangle ::= & \langle identifier \rangle \langle possibleident \rangle \{ \\ & \text{incluir_tipo}(identifier.entrada, identifier-list.tipo) \\ & possibleident.tipo = identifier-list.tipo \} \end{aligned} \quad (7)$$

$$\langle possibleident_1 \rangle ::= ', ' \langle identifier \rangle (8) \langle possibleident \rangle \quad (8)$$

$$| \text{'}\lambda\text{' } \quad (9)$$

$$\langle type \rangle ::= \langle int \rangle \{ type.tipo = int \} \quad (10)$$

$$| \langle string \rangle \{ type.tipo = string \} \quad (11)$$

$$\langle stmt-list \rangle ::= \langle stmt \rangle \langle stmtlist \rangle \quad (12)$$

$$\langle stmtlist \rangle ::= \langle stmt-list \rangle \quad (13)$$

$$| \text{'}\lambda\text{' } \quad (14)$$

$$\langle stmt \rangle ::= \langle assign-stmt \rangle ; \quad (15)$$

$$| \langle if-stmt \rangle \quad (16)$$

$$| \langle while-stmt \rangle \quad (17)$$

$$| \langle read-stmt \rangle ';' \quad (18)$$

$$| \langle write-stmt \rangle ';' \quad (19)$$

$$\langle assign-stmt \rangle ::= \langle identifier \rangle '=' \langle simple-expr \rangle \{ \quad (20)$$

se existir(obter_tipo(identifier.entrada)) então assign-stmt.tipo = void

se obter_tipo(identifier.entrada) = simple_expr.tipo então assign-stmt.tipo = void

senão assign-stmt.tipo = erro }

$$\langle if-stmt \rangle ::= \text{'if' } \langle condition \rangle \text{'then' } \langle stmt-list \rangle \langle if-stmt-prime \rangle \quad (21)$$

{ se condition.tipo = logico então if-stmt.tipo = void

senão if-stmt.tipo = erro }

$$\langle if-stmt-prime \rangle ::= \text{'end' } \quad (22)$$

$$| \text{'else' } \langle stmt-list \rangle \langle end \rangle \quad (23)$$

$$\langle condition \rangle ::= \langle expression \rangle \quad (24)$$

$$\langle while-stmt \rangle ::= \text{'do' } \langle stmt-list \rangle \langle stmt-suffix \rangle \quad (25)$$

$$\langle stmt-suffix \rangle ::= \text{'while' } \langle condition \rangle \text{'end' } \{ \quad (26)$$

se condition.tipo = logico então if-stmt.tipo = void

senão stmt-suffix.tipo = erro }

$$\langle read-stmt \rangle ::= \text{'scan' } '(' \langle identifier \rangle ')' \quad (27)$$

{ se existir(obter_tipo(identifier.entrada)) então read-stmt.tipo = void

senão

read-stmt.tipo = erro

$$\langle write-stmt \rangle ::= \langle print \rangle \text{'('} \langle writable \rangle \text{')}' \quad (28)$$

$$\langle writable \rangle ::= \langle simple-expr \rangle \quad (29)$$

$$| \langle literal \rangle \quad (30)$$

$$\langle expression \rangle ::= \langle simple-expr \rangle \langle expression-prime \rangle \quad (31)$$

$$\{ \text{expression-prime.tipo} = \text{expression.tipo} \}$$

$$\langle expression-prime \rangle ::= \langle relop \rangle \langle simple-expr \rangle \quad (32)$$

$$| \text{'}\lambda\text{' } \quad (33)$$

$$\langle simple-expr \rangle ::= \langle term \rangle \langle simple-expr-prime \rangle \quad (34)$$

{ simple-expr-prime.tipo = term.tipo

se term.tipo = simple-expr-prime.tipo então simple-expr.tipo = term.tipo

senão term.tipo = erro }

$$\langle simple-expr-prime \rangle ::= \langle addop \rangle \langle term \rangle \langle simple-expr-prime \rangle \quad (35)$$

{ se term.tipo = string e simple-expr-prime = string e addop.val = '+' então
simple-expr-prime.tipo = string

se term.tipo = logico e simple-expr-prime.tipo = logico e addop.val = '||' então
simple-expr-prime.tipo = logico

se term.tipo = int e simple-expr-prime.tipo = int e addop.val != '||' então
simple-expr-prime.tipo = int

senão simple-expr-prime.tipo = erro }

$$| \text{'}\lambda\text{' } \quad (36)$$

$$\langle term \rangle ::= \langle factor-a \rangle \langle term-prime \rangle \{ \quad (37)$$

se factor-a.tipo = term-prime.tipo então term.tipo = factor-a.tipo

senão term.tipo = erro }

$$\langle term-prime \rangle ::= \langle mulop \rangle \langle factor-a \rangle \langle term-prime \rangle \quad (38)$$

{ se factor-a.tipo = int e term-prime = int e mulop.val != '' então
term-prime.tipo = int

se factor-a.tipo = logico e term-prime = logico e mulop.val = '' então
term-prime.tipo = logico

senão term-prime.tipo = erro }

$$| \text{'}\lambda\text{' } \quad (39)$$

$$\langle factor-a \rangle ::= \langle factor \rangle \{ \text{factor-a.tipo} = \text{factor.tipo} \} \quad (40)$$

$$| \text{'!'} \langle factor \rangle \{ \text{factor-a.tipo} = \text{factor.tipo} \} \quad (41)$$

$$| \text{'-' } \langle factor \rangle \{ \text{factor-a.tipo} = \text{factor.tipo} \} \quad (42)$$

$$\langle factor \rangle ::= \langle identifier \rangle \{ \quad \quad \quad \} \quad (43)$$

se existir(obter_tipo(identifier.entrada)) então factor.tipo = void
senão factor.tipo = erro }

$$| \quad \langle constant \rangle \{ factor.tipo = constant.tipo \} \quad (44)$$

$$| \quad '(\langle expression \rangle)'\{ factor.tipo = expression.tipo \} \quad (45)$$

$$\langle constant \rangle ::= integer-constant \{ constant.tipo = int \} \quad (46)$$

$$| \quad literal \{ constant.tipo = string \} \quad (47)$$

$$\langle relop \rangle ::= '==' \quad (48)$$

$$| \quad '>' \quad (49)$$

$$| \quad '>=' \quad (50)$$

$$| \quad '<' \quad (51)$$

$$| \quad '<=' \quad (52)$$

$$| \quad '!=' \quad (53)$$

$$\langle addop \rangle ::= '+' \quad (54)$$

$$| \quad '-' \quad (55)$$

$$| \quad '||' \quad (56)$$

$$\langle mulop \rangle ::= '*' \quad (57)$$

$$| \quad '/' \quad (58)$$

$$| \quad '&&' \quad (59)$$

Ações semânticas não apresentadas na gramática:

8 { incluir_tipo(identifier.entrada, possibleident₁.tipo) }

32 { se simple-expr.tipo = string e expression-prime.tipo = string
e (relop.val = '==' ou relop.val = '!=') então
expression-prime.tipo = logico
se simple-expr.tipo = int e expression-prime.tipo = int então
expression-prime.tipo = logico }

2.2 Implementação do analisador semântico

Devido a natureza incremental deste trabalho, optou-se pela utilização da linguagem de programação Java para permitir incrementar as etapas anteriores. As instruções correspondentes ao analisador semântico foram incorporadas aos métodos do analisador sintático descendente na classe *Parser*. Não foi utilizada nenhuma biblioteca externa à linguagem Java.

2.2.1 Arquitetura da implementação

O diagrama de classe gerado no trabalho anterior (Figura 3) foi mantido, visto que somente alterações pontuais foram efetuadas nos métodos da classe *Parser*. A arquitetura do KCompiler baseou-se nas diretrizes e recomendações de (ANDREW; JENS, 2002).

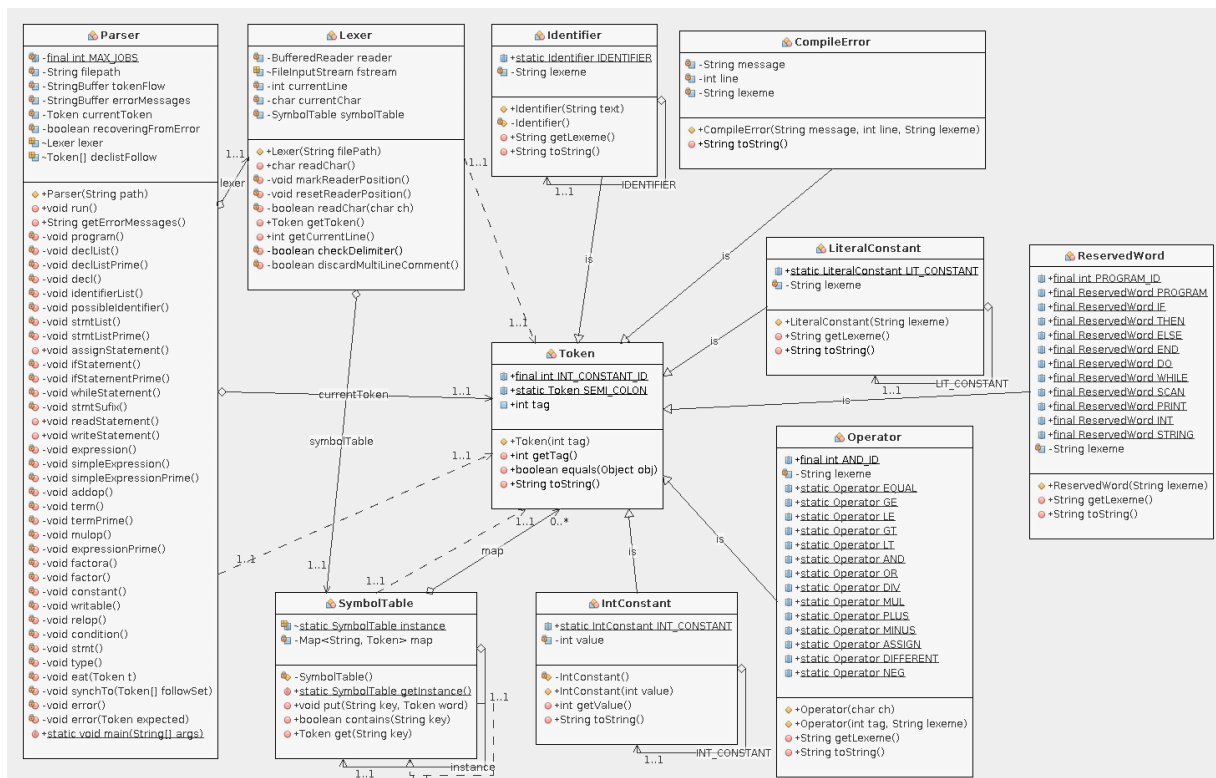


Figura 3 – Diagrama de classe do compilador, as classes utilitárias foram omitidas para facilitar o entendimento.

Nesta etapa do projeto, somente a classe *Parser* foi alterada significativamente. A classe *Type* foi criada para atender às necessidades de retorno a partir dos métodos do *Parser*. Estas classes e funções estão detalhadas a seguir:

- **Parser (analisador sintático):** Em cada método correspondente à um símbolo não-terminal da gramática, foi adicionado código respectivo às ações semânticas

especificadas previamente. Algumas dessas produções passaram a retornar um objeto *Type*. Os métodos adicionais *SemanticError(String message)* e *checkIdentifierUnicity(Token token, Type type)* foram criados para exibir erros semânticos e detectar redeclarações, respectivamente.

- **Type:** contem o dicionário de tipos de dados, entre os quais estão (1) string, (2) int, (3) logico, (4) void e (5) erro. Além disso, possui métodos para obter o valor do tipo e construtor. Cada método correspondente à um símbolo não-terminal da gramática passou a retornar um objeto *Type* para prover informação de tipo durante o retorno entre procedimentos aninhados.

2.2.2 Recursos adicionais

O compilador foi construído em uma arquitetura paralela, que permite que vários arquivos sejam analisados paralelamente pelos analisadores léxico e sintático. No trabalho anterior foi identificada uma falha de sincronização entre as *threads*, que foi informada à orientadora e reparada nessa etapa do trabalho. Testes de exaustão automatizados com sequências de 1000 execuções por teste foram realizados e mostraram que o problema foi corrigido.

3 Instruções de utilização

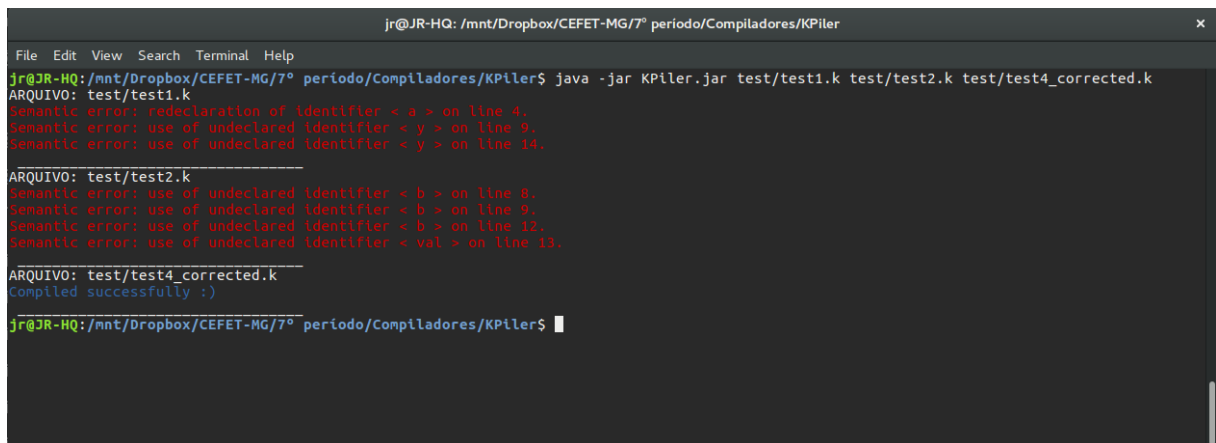
O projeto está preparado para receber o caminho dos arquivos de código-fonte via linha de comando (terminal do Linux ou *prompt* de comandos do Windows). A passagem dos argumentos pode ser configurada no ambiente de desenvolvimento de preferência ou informado na inicialização do arquivo executável .jar ou na execução direta dos arquivos .class. Exemplo de execução:

- **Execução do .jar:** o seguinte comando deve ser executado na mesma pasta do arquivo .jar caso as configurações do projeto anexado sejam mantidas:

```
java -jar KPiler.jar test/test1.k test/test2.k
```

- **Execução dos arquivos .class:** deve-se navegar para a pasta do *build* e executar os seguintes comandos para compilar e executar o código:

```
javac Parser.java  
java Parser test/test1.k test/test2.k
```



```
jr@JR-HQ: /mnt/Dropbox/CEFET-MG/7º período/Compiladores/KPiler  
File Edit View Search Terminal Help  
jr@JR-HQ: /mnt/Dropbox/CEFET-MG/7º período/Compiladores/KPiler$ java -jar KPiler.jar test/test1.k test/test2.k test/test4_corrected.k  
ARQUIVO: test/test1.k  
semantic error: redeclaration of identifier < a > on line 4.  
semantic error: use of undeclared identifier < y > on line 9.  
semantic error: use of undeclared identifier < y > on line 14.  
-----  
ARQUIVO: test/test2.k  
semantic error: use of undeclared identifier < b > on line 8.  
semantic error: use of undeclared identifier < b > on line 9.  
semantic error: use of undeclared identifier < b > on line 12.  
semantic error: use of undeclared identifier < val > on line 13.  
-----  
ARQUIVO: test/test4_corrected.k  
Compiled successfully :)  
jr@JR-HQ: /mnt/Dropbox/CEFET-MG/7º período/Compiladores/KPiler$
```

Figura 4 – Execução do programa via terminal.

4 Avaliação dos resultados

O programa foi submetido a seis testes de compilação. Cinco dos testes foram elaborados pela orientadora deste trabalho e um teste foi elaborado pelo autor. Os arquivos de teste estão disponibilizados em anexo no diretório *test* disponível neste projeto e em sua página do GitHub¹. A página do GitHub se encontra em modo privado até a conclusão da disciplina.

Os testes foram submetidos ao compilador de maneira incremental, isto é, os erros reportados em na fase inicial foram corrigidos e o código submetido novamente de maneira repetida até que não houvessem mais erros. Cumpre observar que os programas-fonte livres de erros léxicos e sintáticos das etapas anteriores foram considerados nesta etapa.

4.1 Teste 1

Código-fonte:

```
1  program
2    int a, b;
3    int result;
4    int a,x,total;
5
6    a = 2;
7    x = 1;
8    scan (b);
9    scan (y);
10   result = (a*b + 1) / 2;
11   print ("Resultado: ");
12   print (result);
13   print ("Total: ");
14   total = y / x;
15   print ("Total: ");
16   print (total);
17 end
```

¹ <<https://github.com/josuerocha/KPiler>>

Resultado da execução 1:

```
1 Semantic error: redeclaration of identifier < a > on line 4.
2 Semantic error: use of undeclared identifier < y > on line 9.
3 Semantic error: use of undeclared identifier < y > on line 14.
```

Os erros apresentados foram corrigidos com as seguintes modificações:

- Remoção da declaração repetida do identificador a na linha 4;
- Declaração do identificador y na linha 4;

O código-fonte submetido ao compilador novamente:

Código-fonte corrigido:

```
1 program
2   int a, b;
3   int result;
4   int x,y,total;
5
6   a = 2;
7   x = 1;
8   scan (b);
9   scan (y);
10  result = (a*b + 1) / 2;
11  print ("Resultado: ");
12  print (result);
13  print ("Total: ");
14  total = y / x;
15  print ("Total: ");
16  print (total);
17 end
```

Resultado da execução 2:

```
1 Compiled successfully :)
```

Nenhum outro erro foi retornado pelo compilador após esta alteração, atendendo aos resultados esperados.

4.2 Teste 2

Código-fonte:

```
1 program
2   int a, c;
3   int d, e;
4   a = 0; d = 35;
5   c = d / 12;
6   scan (a);
7   scan (c);
8   b = a * a;
9   c = b + a * (1 + a*c);
10  print ("Resultado: ");
11  print (c);
12  a = (b + c + d)/2;
13  e = val + c + a;
14  print ("E: ");
15  print (e);
16 end
```

Resultado de execução:

```
1 Semantic error: use of undeclared identifier < b > on line 8.
2 Semantic error: use of undeclared identifier < b > on line 9.
3 Semantic error: use of undeclared identifier < b > on line 12.
4 Semantic error: use of undeclared identifier < val > on line 13.
```

O programa fonte foi corrigido da seguinte maneira:

- Declaração do identificador inteiro b na linha 2;
- Declaração do identificador inteiro val na linha 3.

Código-fonte corrigido:

```
1 program
2   int a, b, c;
3   int d, e, val;
4   a = 0; d = 35;
5   c = d / 12;
6   scan (a);
7   scan (c);
8   b = a * a;
```



```
9      c = b + a * (1 + a*c);
10     print ("Resultado: ");
11     print (c);
12     a = (b + c + d)/2;
13     e = val + c + a;
14     print ("E: ");
15     print (e);
16 end
```

O código-fonte foi submetido novamente ao compilador:

```
1 Compiled successfully :)
```

Nenhum erro foi retornado, atendendo aos resultados esperados.

4.3 Teste 3

Código-fonte:

```
1 program
2   int pontuacao, pontuacaoMaxima, disponibilidade;
3   string pontuacaoMinima;
4   disponibilidade = "Sim";
5   pontuacaoMinima = 50;
6   pontuacaoMaxima = 100;
7   /* Entrada de dados
8   Verifica aprovação de candidatos */
9   do
10    print("Pontuacao Candidato: ");
11    scan(pontuacao);
12    print("Disponibilidade Candidato: ");
13    scan(disponibilidade);
14
15    if ((pontuacao > pontuacaoMinima) && (disponibilidade=="Sim")) then
16      print("Candidato aprovado");
17    else
18      print("Candidato reprovado");
19    end
20  while (pontuacao >= 0) end
21 end
```

Resultado de execução:

```
1 Semantic error: type mismatch on assignment, expected INT received STRING
  on line 4.
2 Semantic error: type mismatch on assignment, expected STRING received INT
  on line 5.
3 Semantic error: incompatible operands in relational expression, type INT
  and STRING on line 15.
4 Semantic error: incompatible operands in relational expression, type INT
  and STRING on line 15.
```

O compilador retornou erros de tipos incompatíveis devido à declaração de variáveis com tipos permutados. A correção foi realizada trocando os tipos dos identificadores `pontuacaoMinima` e `disponibilidade` da seguinte maneira:

- O identificador `disponibilidade` foi declarado com tipo *string* na linha 3.
- O identificador `pontuacaoMinima` foi declarado com tipo *int* na linha 2.

Código-fonte:

```
1 program
2   int pontuacao, pontuacaoMaxima, pontuacaoMinima;
3   string disponibilidade;
4   disponibilidade = "Sim";
5   pontuacaoMinima = 50;
6   pontuacaoMaxima = 100;
7   /* Entrada de dados
8   Verifica aprovação de candidatos */
9   do
10    print("Pontuacao Candidato: ");
11    scan(pontuacao);
12    print("Disponibilidade Candidato: ");
13    scan(disponibilidade);
14
15    if ((pontuacao > pontuacaoMinima) && (disponibilidade=="Sim")) then
16      print("Candidato aprovado");
17    else
18      print("Candidato reprovado");
19    end
20  while (pontuacao >= 0) end
21 end
```

O código-fonte foi submetido novamente:

Resultado de execução:

```
1 Compiled successfully :)
```

O resultado esperado foi alcançado, pois o compilador reportou conclusão com sucesso.

4.4 Teste 4

O código fonte inicial foi submetido ao compilador:

Código-fonte:

```
1 program
2   int a, aux, b;
3   string nome, sobrenome, msg;
4   print(Nome );
5   scan (nome);
6   print("Sobrenome: ");
7   scan (sobrenome);
8   msg = "Ola, " + nome + " " +
9   sobrenome + "!";
10  msg = msg + 1;
11  print (msg);
12  scan (a);
13  scan(b);
14  if (a>b) then
15    aux = b;
16    b = a;
17    a = aux;
18  end
19  print ("Apos a troca: ");
20  print(a);
21  print(b);
22 end
```

Resultado de execução:

```
1 Semantic error: use of undeclared identifier < Nome > on line 4.
2 Semantic error: type mismatch on expression types STRING INT on line 10.
```

O programa fonte apresentou erro na linha 4, pois um identificador não declarado denominado Nome foi informado, possivelmente devido à um erro de digitação.

Além disso, na linha 10 ocorreu uma operação entre uma *string* e um *int*, que é uma operação não permitida pela linguagem. As seguintes correções foram aplicadas:

- O identificador Nome foi alterado para nome na linha 4;
- O segundo operando da instrução na linha 10 foi alterada para uma operação entre *strings*: `msg = msg + "1"`;

Código-fonte corrigido:

```
1 program
2   int a, aux, b;
3   string nome, sobrenome, msg;
4   print(nome );
5   scan (nome);
6   print("Sobrenome: ");
7   scan (sobrenome);
8   msg = "Ola, " + nome + " " +
9   sobrenome + "!";
10  msg = msg + "1";
11  print (msg);
12  scan (a);
13  scan(b);
14  if (a>b) then
15      aux = b;
16      b = a;
17      a = aux;
18  end
19  print ("Apos a troca: ");
20  print(a);
21  print(b);
22 end
```

Resultado de execução:

```
1 Compiled successfully :)
```

A execução atendeu aos resultados esperados não retornando outros erros.

4.5 Teste 5

Código-fonte:

```
1  program
2    int a, b, c, maior, outro;
3
4    do
5      print("A");
6      scan(a);
7      print("B");
8      scan(b);
9      print("C");
10     scan(c);
11     //Realizacao do teste
12     if ( (a>b) && (a>c)) then
13       maior = a;
14     else
15       if (b>c) then
16         maior = b;
17       else
18         maior = c;
19     end
20   end
21   print("Maior valor:");
22   print (maior);
23   print ("Outro? ");
24   scan(outro);
25   while (outro >= 0) end
26 end
```

Resultado de execução:

```
1 Compiled successfully :)
```

Pode-se observar que nenhum erro foi retornado para o este código-fonte.

4.6 Teste 6

Código-fonte:

```
1  /*THIS IS A MULTIPLE LINE COMMENT
2   Josu Rocha Lima */
3
4  program
5   //this is an unclosed string literal
6   string str;
7   int i;
8   int maxtam;
9
10  maxtam = 20;
11  str = "HELLO WORLD";
12  i = 0;
13
14
15  do
16    print(i);
17    print(k); // BUG HERE
18    i = i + 1;
19  while i < maxtam end
20
21  end
```

Resultado de execução:

```
1 Semantic error: use of undeclared identifier < k > on line 17.
```

O código foi corrigido ao eliminar a referência ao identificador não declarado k:

Código-fonte corrigido:

```
1  /*THIS IS A MULTIPLE LINE COMMENT
2   Josu Rocha Lima */
3
4  program
5   //this is an unclosed string literal
6   string str;
7   int i;
8   int maxtam;
9
10  maxtam = 20;
```

```
11     str = "HELLO WORLD";
12     i = 0;
13
14
15     do
16         print(i);
17         i = i + 1;
18     while i < maxtam end
19
20 end
```

O código foi compilado novamente e retornou o seguinte resultado:

```
1 Compiled successfully :)
```

O programa fonte não apresentou erros, correspondendo aos resultados esperados.

5 Conclusão

Neste trabalho relatou-se a experiência dos autores durante a implementação do analisador semântico, uma etapa incremental aos analisadores léxico e sintático.

Percebeu-se a aplicabilidade de esquemas de tradução dirigida por sintaxe em conjunto com o analisador sintático recursivo descendente para mitigar a complexidade do problema de compilação ([AHO; SETHI; ULLMAN, 2007](#)).

Todos os resultados dos testes do analisador semântico do *KPiler* atenderam aos resultados desejados, quanto à precisão dos resultados e tempo de execução.

Referências

AHO, A. V.; SETHI, R.; ULLMAN, J. D. *Compilers: principles, techniques, and tools*. [S.l.]: Addison-wesley Reading, 2007. v. 2.

ANDREW, W. A.; JENS, P. *Modern compiler implementation in Java*. [S.l.]: Cambridge, 2002.