Disciplina: 12037 – Compiladores
Profa. Valéria

Trabalho Prático

Implementação de um compilador para a linguagem Rascal

Especificação geral

O presente trabalho propõe a construção de um compilador simples para uma linguagem experimental denominada **Rascal** (*Reduced Pascal*). Essa linguagem consiste num subconjunto reduzido da linguagem Pascal, com algumas modificações em sua sintaxe original. O compilador deverá realizar as três etapas de análise: léxica, sintática e semântica. Após às análises, o compilador deverá gerar código intermediário para ser interpretado por uma máquina denominada MEPA (Kowaltowski, 1983). Uma implementação dessa máquina será disponibilizada junto com os arquivos de apoio ao trabalho.

O trabalho poderá ser desenvolvido nas seguintes linguagens: C, C++ ou Python. Para auxiliar na implementação das etapas de análise, podem ser utilizadas ferramentas de geração automática de *scanners* e *parsers*, como Flex[1], Bison[2] (para linguagem C/C++) e PLY[3] (para linguagem Python).

É requisito deste trabalho que uma Árvore Sintática Abstrata (AST) seja produzida a partir da análise sintática. O objetivo de se construir uma AST explícita em vez de utilizar as ações semânticas do *parser* para a Tradução Dirigida por Sintaxe (SDT) é que, dessa forma, o código ficará mais organizado (modularizado), facilitando sua manutenção e depuração.

O compilador deverá ser executado a partir da linha de comando, passando como argumentos o arquivo de código-fonte em Rascal e o nome do arquivo de saída, onde será gravado o código-objeto da MEPA.

Normas para desenvolvimento e entrega do trabalho

O trabalho deverá ser desenvolvido **por uma equipe de duas pessoas**. Caso você não consiga formar uma dupla, converse com a professora antecipadamente.

A submissão do trabalho deverá ser feita exclusivamente pelo *Classroom*. **O prazo final para a entrega será até às 23h59min do dia 07/12/2025**.

Deverão ser entregues os seguintes artefatos:

- **Código-fonte** do compilador (compactado no caso de vários arquivos separados);
- **Relatório técnico** escrito em português, de maneira sucinta e objetiva, contendo:
  - Decisões de projeto e de implementação, com justificativas;
  - Visão geral dos módulos e organização do analisador léxico, sintático e semântico;
  - Passo a passo para compilar/executar o compilador;
  - Descrição e justificativas de possíveis etapas não cumpridas durante o desenvolvimento.

[1] https://westes.github.io/flex/manual/
[2] https://www.gnu.org/software/bison/manual/
[3] https://ply.readthedocs.io/en/latest/index.html

Avaliação do trabalho

O código-fonte do compilador será avaliado de acordo com sua organização, correção e requisitos atendidos. Para isso, um conjunto de programas escritos em Rascal será utilizado para verificar as etapas que foram cumpridas no desenvolvimento do compilador.

Também fará parte da avaliação do trabalho, uma apresentação sobre sua implementação, a ser realizada pela equipe nos dias 08/12/2025 e 10/12/2025, conforme agendamento prévio.

Características da linguagem

Por se tratar de um subconjunto de Pascal[4], Rascal compartilha muitas das características dessa linguagem, mas com algumas alterações que serão descritas no decorrer desta especificação. Uma delas é que **Rascal é *case-sensitive***, diferentemente da linguagem Pascal original.

Palavras reservadas

`program`	`begin`	`false`	`if`	`write`
`procedure`	`end`	`true`	`then`	`and`
`function`	`integer`	`while`	`else`	`or`
`var`	`boolean`	`do`	`read`	`not`
				`div`

Símbolos especiais

`(`	`=`	`<=`	`+`	`:=`
`)`	`<>`	`>`	`-`	`:`
`;`	`<`	`>=`	`*`	`,`
				`.`

Rascal **suporta apenas números inteiros.** Desse modo, assim como em Pascal, o operador de divisão inteira é representado pela palavra reservada `div`. Como não trabalharemos com números reais, este será o único operador de divisão disponível.

Tipagem

A linguagem Rascal possui apenas dois tipos primitivos:

- **`integer`**: representa valores numéricos inteiros;
- **`boolean`**: representa valores lógicos (`false` e `true`).

Diferentemente de Pascal, Rascal **não permite** que novos tipos sejam definidos pelo usuário.

Interfaces de entrada e saída

Dois comandos de entrada e saída estão disponíveis em Rascal: `read` e `write`. O primeiro deles solicita a leitura de dados pela interface de entrada padrão; já o segundo exibe valores na interface de saída padrão.

[4] Este livro apresenta uma visão geral da linguagem Pascal: https://www.ime.usp.br/~slago/slago-pascal.pdf

Eles são procedimentos pré-definidos pela linguagem e são capazes de receber um número indeterminado de argumentos do tipo `integer` ou `boolean`. No entanto, não é possível "escrever uma mensagem" associada à etapa de leitura, visto que Rascal não suporta o tipo *string*. Seguem alguns exemplos de uso desses comandos:

```
read(x, y);
write(x, y, 2*x+y, 3);
```

Especificação léxica

Identificadores

Os identificadores são nomes criados pelo usuário e associados a entidades do programa, como variáveis, sub-rotinas (funções e procedimentos), etc. Os identificadores devem ser formados apenas por letras (minúsculas ou maiúsculas), sublinhas '_' ou dígitos (0 a 9), devendo iniciar com uma letra. Nas regras gramaticais da linguagem, o símbolo `id` será utilizado para expressar um identificador qualquer.

Números

As constantes numéricas devem ser representadas na base decimal e podem conter qualquer combinação de dígitos entre 0 e 9. Os números negativos não serão processados na fase léxica, mas sim na fase sintática. Dessa forma, o número −42, por exemplo, consiste em dois símbolos: '−' e '42', representando uma expressão de negação numérica (inversão de sinal) sobre a constante 42.

Lógicos

As constantes lógicas falso e verdadeiro são representadas respectivamente pelas palavras reservadas **`false`** e **`true`**.

Comentários e símbolos a serem ignorados

A linguagem Rascal não aceita a inserção de comentários no código-fonte. Além disso, espaços em branco, tabulações e quebras de linha não possuem nenhum significado específico e devem ser ignorados.

Especificação sintática

A gramática de Rascal, disponível no **Anexo I** desta especificação, foi adaptada a partir da gramática apresentada por Kowaltowski (1983), a qual já é um subconjunto da linguagem Pascal. No entanto, caso sua equipe for utilizar alguma ferramenta auxiliar para implementação do compilador, certifique-se de que a notação utilizada para representação da gramática está de acordo com a sintaxe aceita pela ferramenta a ser utilizada.

Por exemplo, a gramática do Anexo I utiliza a notação EBNF, a qual não corresponde ao formato aceito pelo Bison ou pelo PLY, especialmente pelas construções:

- { α } para indicar repetição; e

- [α] para indicar opção.

No entanto, essas construções podem ser facilmente transformadas em construções equivalentes da seguinte forma:

- Regras de repetição na forma A → β { α }, se transformam em A → Aα | β
- Regras opcionais na forma A → β [α], se transformam em A → β | βα

Você vai precisar dessas transformações principalmente para as regras de formação de lista e de expressões.

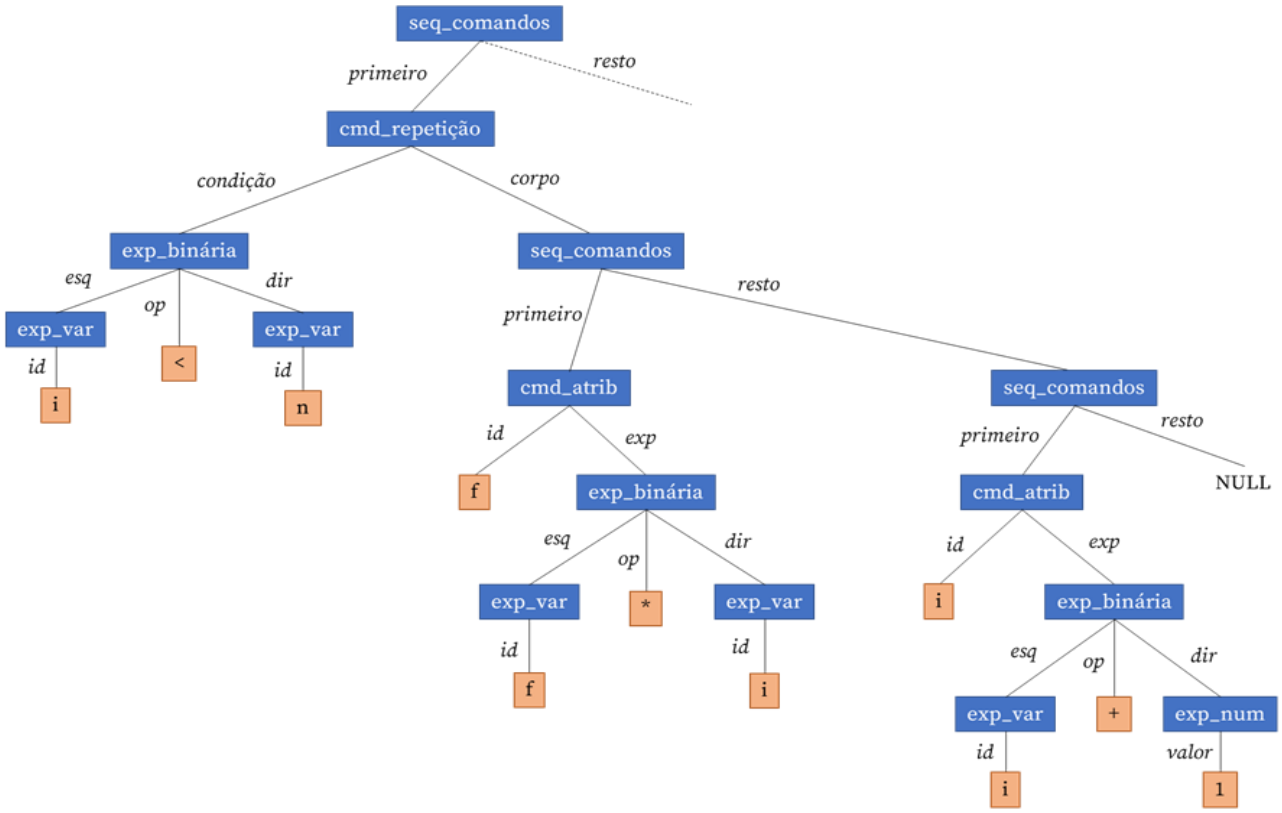
Construção da Árvore Sintática Abstrata

Como dito anteriormente, a análise sintática terá como responsabilidade, além de verificar a correção da sintaxe do programa e emitir erros apropriados, construir uma Árvore Sintática Abstrata (AST).

A AST é uma representação simplificada da estrutura do programa, abstraindo as derivações sintáticas em nós que representam as construções da linguagem, em vez dos símbolos propriamente ditos. Por exemplo, considere o seguinte fragmento de código:

```
while i < n do
begin
  f := f * i;
  i := i + 1
end;
```

A AST desse fragmento poderia ser construída da seguinte forma:



Para uma inspiração de como definir as estruturas que irão compor a AST, investigue os códigos disponibilizados por Appel e Ginsburg (1998)[5], principalmente os que se referem à "Abstract Syntax".

Especificação semântica

Após a construção da AST, a análise semântica e a geração de código podem ser realizadas por meio de uma travessia na árvore. De maneira geral, cada nó da AST terá suas próprias regras de verificação semântica e geração de código. A seguir, são apresentados os tratamentos semânticos das principais construções da linguagem.

Programa

- Um programa é composto por declarações de variáveis (opcionais), sub-rotinas (opcionais) e uma sequência obrigatória de comandos.
- Como ele é a estrutura principal (nó raiz da AST), a análise inicia e termina no programa. Nesse ponto, a tabela de símbolos é criada e o escopo global é estabelecido.
- Todas as declarações realizadas no programa estão dentro do escopo global, que permanece ativo até o fim da execução.
- Cada sub-rotina define seu próprio escopo local e, como não são permitidas declarações em blocos de comandos, blocos `begin ... end` não criam novo escopo.
- O identificador do programa deve ser instalado na tabela de símbolos como sendo da categoria "programa".

Declaração

- Declarações de variáveis, funções, procedimentos e parâmetros instalam os símbolos correspondentes e suas vinculações (tipo, escopo, posição no escopo etc.) na tabela de símbolos.
- Antes de instalar um novo identificador, deve-se verificar se ele já existe no mesmo escopo.
    - Caso não exista, ele é instalado normalmente com suas vinculações.
    - Caso exista, a nova declaração é desconsiderada e deve ser emitido um alerta.

Comandos

Os comandos que terão verificação semântica são:

- **Atribuição:**
    - A variável à esquerda da atribuição deve estar declarada e visível no escopo atual.
    - A expressão à direita deve ser semanticamente válida e possuir o mesmo tipo da variável à esquerda.
- **Condicional e repetição:**
    - A expressão condicional dos comandos **`if`** e **`while`** devem ser válidas e resultar em um valor do tipo lógico.

[5] https://www.cs.princeton.edu/~appel/modern/c/project.html

- **Chamada de procedimento:**
  - O procedimento deve estar declarado e visível no escopo atual.
  - O número de argumentos deve coincidir com o número de parâmetros.
  - Os tipos dos argumentos devem corresponder, na mesma ordem, aos tipos dos parâmetros.
  - Todos os parâmetros são passados por valor, i.e., os argumentos reais são avaliados e copiados para os parâmetros formais.
- **Retorno de função:**
  - Toda função deve retornar exatamente um valor.
  - O tipo do valor retornado deve coincidir com o tipo declarado na função.
  - O retorno é realizado por meio de atribuição ao próprio identificador da função dentro de seu corpo.
  - A ausência de retorno, ou retorno de tipo incompatível, deve gerar erro semântico.
- **Leitura (`read`):**
  - Os argumentos devem ser variáveis declaradas e visíveis no escopo atual.
- **Escrita (`write`):**
  - Os argumentos devem ser expressões válidas e bem tipadas.
- **Expressões:**
  - **Aritmética:** O(s) operando(s) devem ser do tipo inteiro. O tipo resultante é inteiro.
  - **Relacional**: Os operandos devem ser do tipo inteiro. O tipo resultante é lógico.
  - **Igualdade/Diferença**: Os operandos devem ser do mesmo tipo primitivo. O tipo resultante é lógico.
  - **Lógica**: O(s) operando(s) devem ser do tipo lógico. O tipo resultante é lógico
  - **Uso de variável**: A variável deve estar declarada e visível no escopo atual. O tipo resultante do uso é o tipo declarado para a variável.
  - **Chamada de Função**: Análoga à chamada de procedimento. O tipo resultante da chamada é o tipo declarado para a função.

Geração de código intermediário para a máquina MEPA

A especificação do código intermediário, bem como da máquina MEPA, será apresentada em sala de aula e constarão no material de aula a ser disponibilizado.

Casos omissos

Quaisquer casos relacionados ao desenvolvimento deste trabalho que não foram esclarecidos nesta especificação deverão ser arguidos diretamente com a professora.

Referências

APPEL, A. W.; GINSBURG, M. **Modern Compiler Implementation in C**. Cambridge: Cambridge University Press, 1998.

KOWALTOWSKI, T. **Implementação de Linguagens de Programação**. Rio de Janeiro: Guanabara Dois, 1983.

Anexo I - Gramática da linguagem Rascal

Regras léxicas para identificadores e números

```
<número> ::= <dígito> { <dígito> }

<dígito> ::= 0-9

<identificador> ::= <letra> { <letra> | <dígito> | '_'}

<letra> ::= a-zA-Z
```

Regras de produção da gramática

```
<programa> ::=
    'program' <identificador> ';' <bloco> '.'

<bloco> ::=
    [ <seção_declaração_variáveis> ]
    [ <seção_declaração_subrotinas> ]
    <comando_composto>

<seção_declaração_variáveis> ::=
    'var' <declaração_variáveis> ';' { <declaração_variáveis> ';' }

<declaração_variáveis> ::=
    <lista_identificadores> ':' <tipo>

<lista_identificadores> ::=
    <identificador> { ',' <identificador> }

<tipo> ::=
    'boolean' | 'integer'

<seção_declaração_subrotinas> ::=
    { ( <declaração_procedimento> | <declaração_função> ) ';' }

<declaração_procedimento> ::=
    'procedure' <identificador> [ <parâmetros_formais> ] ';' <bloco_subrot>

<declaração_função> ::=
    'function' <identificador> [ <parâmetros_formais> ] ':' <tipo> ';'
<bloco_subrot>

<bloco_subrot> ::=
    [ <seção_declaração_variáveis> ]
    <comando_composto>
```

```
<parâmetros_formais> ::=
    '(' <declaração_parâmetros> { ';' <declaração_parâmetros> } ')'

<declaração_parâmetros> ::=
    <lista_identificadores> ':' <tipo>

<comando_composto> ::=
    'begin' <comando> { ';' <comando> } 'end'

<comando> ::=
      <atribuição>
    | <chamada_procedimento>
    | <condicional>
    | <repetição>
    | <leitura>
    | <escrita>
    | <comando_composto>

<atribuição> ::=
    <identificador> ':=' <expressão>

<chamada_procedimento> ::=
    <identificador> [ '(' <lista_expressões> ')' ]

<condicional> ::=
    'if' <expressão> 'then' <comando> [ 'else' <comando> ]

<repetição> ::=
    'while' <expressão> 'do' <comando>

<leitura> ::=
    'read' '(' <lista_identificadores> ')'

<escrita> ::=
    'write' '(' <lista_expressões> ')'

<lista_expressões> ::=
    <expressão> { ',' <expressão> }

<expressão> ::=
    <expressão_simples> [ <relação> <expressão_simples> ]

<relação> ::=
    '=' | '<>' | '<' | '<=' | '>' | '>='

<expressão_simples> ::=
    <termo> { ( '+' | '-' | 'or' ) <termo> }
```

```
<termo> ::=
    <fator> { ( '*' | 'div' | 'and' ) <fator> }

<fator> ::=
      <variável>
    | <número>
    | <lógico>
    | <chamada_função>
    | '(' <expressão> ')'
    | 'not' <fator>
    | '-' <fator>

<variável> ::=
    <identificador>

<lógico> ::=
      'false'
    | 'true'

<chamada_função> ::=
    <identificador> [ '(' <lista_expressões> ')' ]
```