Universidade Estadual de Maringá
Centro de Tecnologia
Departamento de Informática

Disciplina: 9797/2 – Compiladores

Professor: Lucas P. Nanni

Especificação do Trabalho

O presente trabalho propõe a construção de um compilador muito simples para uma linguagem proposta pelo professor, chamada Rascal (Reduced Pascal). Essa linguagem é subconjunto bastante reduzido da linguagem Pascal, com pequenas modificações em sua sintaxe original. O compilador deverá gerar código para uma máquina também muito simples, chamada MEPA (Kowaltowski, 1983). Uma implementação desta máquina, desenvolvida pelo Prof. Bruno Müller Junior (UFPR), será disponibilizada junto com os arquivos de apoio ao trabalho.

Para o desenvolvimento do trabalho, recomenda-se a utilização da linguagem C ou C++, e das ferramentas Flex[1] e Bison[2] para a produção dos analisadores léxico e sintático, respectivamente. Caso prefira utilizar outra linguagem e/ou ferramentas, entre em contato com o professor com antecedência para que a elas sejam avaliadas.

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 Bison para a Tradução Dirigida por Sintaxe (SDT) é que, dessa forma, o código ficará mais organizado (modularizado), facilitando sua manutenção e depuração.

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

Normas de realização, entrega e avaliação do trabalho

O trabalho poderá ser desenvolvido em **equipe de no máximo dois alunos** e deverá ser entregue até o dia **01/12/2021** pelo Classroom.

A entrega do trabalho deverá conter:

- Código-fonte do compilador
- Relatório sucinto e objetivo contendo:
    - Decisões de projeto e de implementação, com justificativas;
    - Etapas cumpridas e não cumpridas no desenvolvimento;
    - Visão geral dos módulos e organização do analisador léxico e sintático;

[1] https://westes.github.io/flex/manual/
[2] https://www.gnu.org/software/bison/manual/

- Passo a passo para compilar/executar o compilador.

Avaliação do trabalho

O código do compilador será avaliado mediante sua organização, correção e requisitos atendidos. Para isso, um conjunto de programas escritos em Rascal será utilizado pelo professor 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 01/12/2021 e 06/12/2021, conforme agendamento prévio pelo professor.

Características da linguagem

Por se tratar de um subconjunto de Pascal, Rascal compartilha a maioria das características dessa linguagem, mas com pequenas alterações que serão descritas no decorrer desta especificação.

Palavras reservadas

Rascal possui as seguintes palavras reservadas:

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

Símbolos

Rascal possui os seguintes símbolos:

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

Assim como em Pascal, o operador de divisão inteira é representado pela palavra **div**. Como não trabalharemos com números reais, este será o único operador de divisão disponível.

Sistemas de tipos

A linguagem possui apenas dois tipos primitivos: **integer**, o qual representa valores numéricos inteiros, e **boolean**, o qual representa valores lógicos (**false** e **true**). Diferente de Pascal, Rascal não permite que novos tipos sejam definidos pelo usuário.

Entrada e Saída

Dois comandos de entrada e saída estão disponíveis: **read** e **write**. Eles são procedimentos pré-definidos pela linguagem e são capazes de receber um número indeterminado de argumentos do tipo **integer** ou **boolean**.

Exemplo:

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

Especificação Léxica

Identificadores

Os identificadores são nomes associados a entidades do programa, como variáveis, tipos, sub-rotinas (funções e procedimentos), etc. Estes nomes podem ser criados pelo usuário, mas também podem ser pré-definidos pela linguagem.

Em Rascal, os nomes dos dois tipos primitivos (**integer** e **boolean**) são pré-definidos pela linguagem. Dessa forma, eles não são palavras reservadas e deverão ser instalados na tabela de símbolos logo no início da fase de análise semântica / tradução. Caso estes identificadores sejam utilizados durante o programa, deve-se verificar a correção de seu uso. Por exemplo, o identificador **integer**, que foi instalado na tabela de símbolos como um símbolo da categoria "*tipo*", não pode ser empregado como nome de uma nova variável declarada pelo usuário.

Os identificadores devem ser formados apenas por letras (minúsculas ou maiúsculas), sublinhas '_' ou dígitos (0 a 9), sendo que devem 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

Há dois tipos de comentários:

- Comentário de linha: inicia em qualquer local do programa com o símbolo **//** e se estende até o final da linha.
- Comentário de bloco: inicia em qualquer local do programa com o símbolo **{** e se estende até o primeiro símbolo **}**, o qual pode ocorrer na mesma linha ou em alguma linha posterior.

De forma geral, os comentários podem conter qualquer tipo de símbolo, inclusive os não permitidos pela linguagem. Os comentários devem ser processados corretamente pelo analisador léxico e em seguida descartados. Caso um comentário de bloco não seja terminado até o final do arquivo, um aviso deve ser emitido pelo analisador léxico.

Especificação Sintática

A gramática de Rascal, disponível no Anexo I, foi adaptada pelo professor a partir da gramática apresentada por (Kowaltowski, 1983), a qual já é um subconjunto da linguagem Pascal. Entretanto, a gramática de Rascal ainda não está em sua versão final, exigindo que você realize algumas modificações.

Modificação 1

A primeira modificação visa simplificar ainda mais a linguagem, tornando o bloco de sub-rotinas mais restrito que o bloco de programa, sem a seção de declaração de sub-rotinas. Isso impossibilitará a declaração de sub-rotinas aninhadas, reduzindo a complexidade do compilador.

Modificação 2

Outra modificação se refere ao separador dos comandos dentro de um comando composto. Em Pascal, os comandos inseridos entre **begin** e **end** devem ser separados por **;**. Este símbolo somente deve ser empregado *entre* os comandos, fazendo com que o último comando dentro do comando composto não seja seguido de **;**.

Exemplo:

```
begin
    x := 2 * a + b;
    y := x div 2;
    write(x, y)
end
```

O professor acredita que o programador, por costume, coloque um **;** após o último comando (assim como ele faz em todos os outros comandos), ao invés de se lembrar que deve omiti-lo. Muitas implementações do Pascal resolvem isso empregando a existência de um comando "vazio" que será produzido logo após o último comando caso o programador coloque um **;** ao final dele. Entretanto, Rascal deve superar este problema ao exigir que um **;** seja colocado *após* cada comando inserido em um comando composto, assim como acontece nas linguagens C e Java, por exemplo.

Modificação 3

Por fim, a gramática apresentada não está em um formato aceito pelo Bison, especialmente pelas construções:

- { α } para indicar repetição;
- [α] para indicar opção;

Essas construções podem ser facilmente transformadas em construções equivalentes da seguinte forma:

- Regras de repetição na forma $A \rightarrow \beta \{ \alpha \}$, se transformam em $A \rightarrow A\alpha \mid \beta$
- Regras opcionais na forma $A \rightarrow \beta [\alpha]$, se transformam em $A \rightarrow \beta \mid \beta\alpha$

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

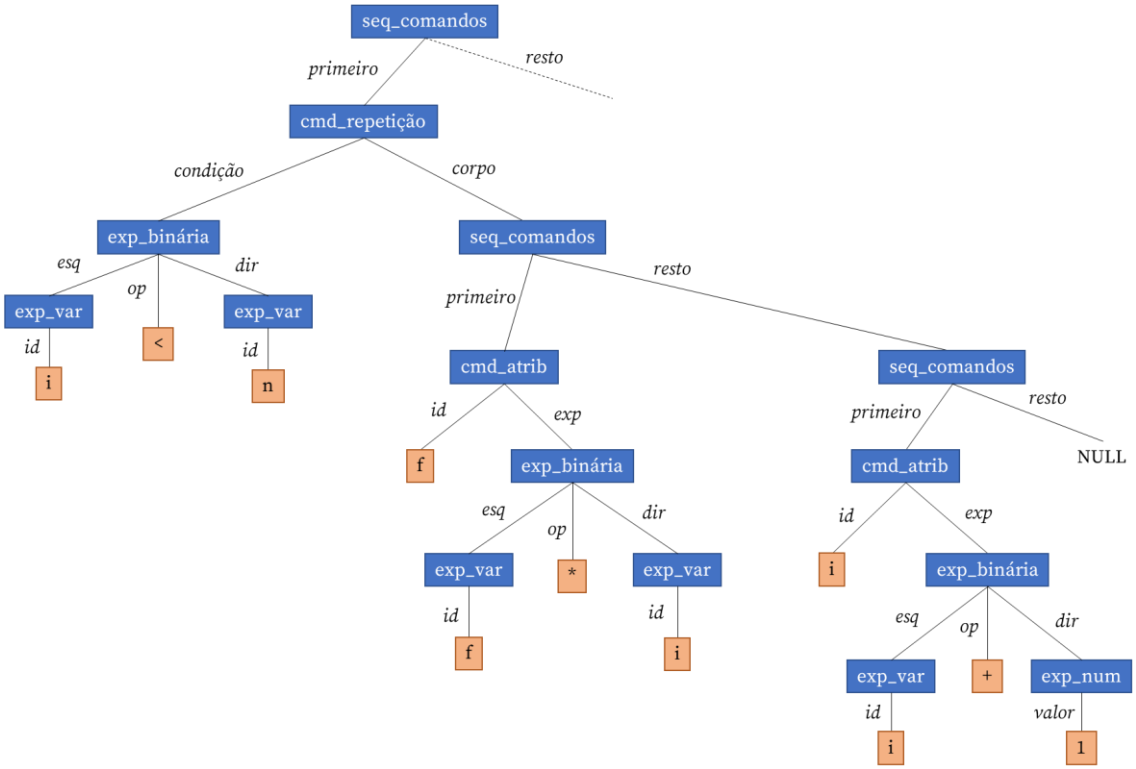
Construção da Árvore Sintática Abstrata

Como comentado 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[3] por (Appel & Ginsburg, 1998), principalmente os que se referem a “Abstract Syntax”.

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

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 consiste em uma sequência de declarações de variáveis (opcional), seguida de uma sequência de declarações de sub-rotinas (opcional) e, por fim, uma sequência de comandos (mandatória). Por ser a estrutura principal do programa (nó raiz da AST), a análise começa e termina com ela. Portanto, é aqui que a tabela de símbolos será inicializada e o escopo global será definido. Todas as declarações realizadas no programa estão dentro do escopo global, respeitando a ordem de declaração. O identificador do programa deve ser instalado na tabela de símbolos como sendo da categoria “programa”.

Declaração

A declaração de variáveis, funções e procedimentos, bem como de seus parâmetros, é responsável por adicionar os símbolos envolvidos e suas vinculações (tipo, escopo, posição no escopo etc.) na tabela de símbolos. De maneira geral, a cada declaração, deve-se primeiro consultar a tabela de símbolos para verificar se o respectivo identificador já não foi instalado no escopo atual. Caso não tenha sido, ele é instalado com suas vinculações adequadas. Caso já tenha sido, um alerta deve ser emitido e a nova declaração deve ser desconsiderada.

Comandos

Os comandos que merecem verificação semântica são:

Atribuição

- A variável à esquerda da atribuição deve estar declarada e acessível no escopo atual. A expressão à direita deve ser semanticamente correta e possuir o mesmo tipo da variável à esquerda.

Condicional (if) e repetição (while)

- A expressão condicional do **if** e do **while** deve ser válida e resultar em um valor do tipo lógico.

Chamada de Procedimento

- O procedimento deve estar declarado e visível no escopo atual.
- O número de argumentos fornecidos deve ser o mesmo de parâmetros encontrados na declaração do procedimento.
- Os argumentos fornecidos devem ter a mesma ordem de tipo utilizada na declaração do procedimento. Além disso, no caso de parâmetros de passagem

por referência, o argumento deve ser uma variável (expressão de uso de variável).

Expressões

Aritmética (+, -, *, div)

- 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 (and, or, not)

- 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

- Análise análoga à chamada de procedimento. O tipo resultante da chamada é o tipo declarado para a função.

Referências

Appel, A. W., & Ginsburg, M. (1998). *Modern Compiler Implementation in C.* Cambridge: Cambridge University Press.

Kowaltowski, T. (1983). *Implementação de Linguagens de Programação.* Rio de Janeiro: Guanabara Dois.

Anexo I

A Gramática de Rascal.

```
<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> ::= <identificador>

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

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

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

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

<declaração_parâmetros> ::=
    [ 'var' ] <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> ')' ]

/* as regras abaixo são de responsabilidade do analisador léxico,
   mas foram descritas aqui para completar a gramática.
*/

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

<dígito> ::= 0-9

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

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