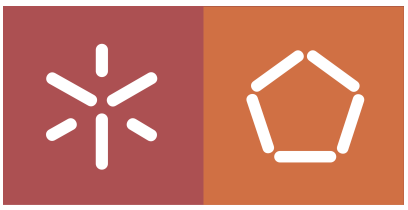

Universidade do Minho
Escola de Engenharia

Compilador para Pascal Standard

Projeto de Processamento de Linguagens

Membros do Grupo
- Afonso Sousa - a104262
- Edgar Pinto - a104081
- Luís França - a104259

Ano Letivo: 2024/2025

Índice

Introdução

O presente projeto consiste na implementação de um compilador para a linguagem Pascal, desenvolvido no âmbito da unidade curricular de Processamento de Linguagens. Este compilador tem como objetivo a tradução de programas escritos em Pascal para um código intermédio e, posteriormente, é gerado o código-máquina correspondente para ser executado na máquina virtual disponibilizada pela equipa docente.

O compilador implementa as fases tradicionais de um processo de compilação, nomeadamente:

- **Análise Léxica**: Responsável por converter o código fonte em tokens
- **Análise Sintática**: Encarregue da construção da árvore sintática abstrata (AST)
- **Análise Semântica**: Efetua a verificação de tipos e estrutura do programa

- **Geração de Código**: Produz o código intermédio para a máquina virtual alvo

A implementação foi realizada em Python, utilizando a biblioteca `ply` (Python Lex-Yacc) para as fases de análise léxica e sintática. O compilador suporta as principais estruturas de controlo do Pascal (if-then-else, for, while), operações aritméticas, variáveis, arrays e chamadas a funções como writeln e readln.

Arquitetura

O compilador está estruturado de forma modular, dividindo as responsabilidades em diferentes componentes:

1. tokenizer.py: Implementa o analisador léxico utilizando o módulo `lex` da biblioteca `ply`. Define os tokens da linguagem, incluindo palavras-chave, operadores, identificadores e literais.
2. pascal_yacc.py: Contém o analisador sintático que utiliza a gramática definida para construir a árvore sintática abstrata (AST). Cada regra de produção corresponde a uma classe em node.py.
3. node.py: Define as classes que representam os nós da AST. Cada classe contém um método generate() que produz o código da máquina virtual correspondente.
4. symbol_table.py: Implementa a tabela de símbolos que armazena informações sobre variáveis, arrays e as suas posições na stack da máquina virtual.
5. main.py: Programa principal que coordena o processo de compilação, lendo o código fonte Pascal, invocando o parser e gerando o código de máquina num ficheiro de saída.

O fluxo de execução do compilador começa com a leitura do código fonte, passa pela análise léxica e sintática, constrói a AST, realiza a análise semântica através da tabela de símbolos e finalmente gera o código da máquina virtual.

Análise Léxica

O módulo tokenizer.py implementa o analisador léxico utilizando o `lex`, como referido anteriormente. Este componente é responsável por dividir o código fonte em tokens reconhecíveis pelo analisador sintático.

Tokens Implementados:

- Palavras-chave: program, var, begin, end, if, then, else, while, for, to, downto, etc.
- Tipos de dados: integer, boolean, char, real, string, array
- Operadores: aritméticos (+, -, *, /), relacionais (<, >, <=, >=, =, <>), lógicos (and, or, not)
- Símbolos especiais: parêntesis, parêntesis rectos, ponto, vírgula, ponto-e-vírgula, etc.
- Identificadores: nomes de variáveis seguindo a convenção do Pascal
- Literais: números inteiros, números reais, cadeias de caracteres, caracteres, valores booleanos

Tratamento Especial:

- Comentários: Ignorados pelo lexer, tanto os delimitados por {...} como por (...)
- Espaços em branco: Ignorados pelo lexer
- Strings e caracteres: Extraídos sem as aspas delimitadoras

- Case-insensitive: O lexer foi configurado para tratar maiúsculas e minúsculas como equivalentes
- Cada token é definido através de expressões regulares ou funções específicas, permitindo o reconhecimento preciso de cada elemento da linguagem Pascal.

Análise Sintática

A análise sintática é implementada no módulo pascal_yacc.py utilizando a biblioteca PLY (yacc). Este componente usa os tokens fornecidos pela análise léxica para construir uma árvore sintática abstrata (AST).

Definição da gramática em BNF: Gramática

Gramática Implementada

A gramática implementada suporta:

- Estrutura básica de um programa Pascal: cabeçalho, declarações de variáveis e bloco de comandos
- Declarações de variáveis: tipos simples e arrays
- Expressões aritméticas e lógicas: com precedência correcta de operadores
- Estruturas de controlo: if-then-else, while, for
- Chamadas de função: principalmente para entrada e saída (readln, writeln)
- Arrays: declaração, acesso a elementos e atribuição

O axioma desta gramática é a produção `program`.

Da gramática implementada queremos destacar alguns casos particulares:

Recursividade de listas

Para definir listas de valores tivemos atenção de aplicar recursividade à esquerda pelo facto do yacc ser um parser LALR.

Como exemplo demosntramos a declaração das produuções var_declarations e var_declaration que aplicam esta estratégia; as restantes produções que involvem listas (identifiers_list, args_list, etc.) seguem a mesma lógica

```
<var_declarations> -> <var_declarations> <var_declaration>
                    | <var_declaration>

<var_declaration> -> <identifiers_list> ":" <type> ";"
```

Blocos If-Then-Else

Para definir blocos if then else aplicamos as seguintes produções:

```
<if> -> "IF" <expressionBool> "THEN" <command_list> <else>

<else> -> "ELSE" <command_list>
        | ε
```

Estas produções primeiro identificam a condição do if através da produção expressionBool e depois a lista de comandos através da produção command_list a existência ou não de um segmento else depende da seguinte produção que ou dá match com o terminal ELSE ou com empty.

Apesar de corretas, essas produções podem gerar um conflito shift-reduce durante a análise sintática. No entanto, por padrão, o Yacc resolve esse tipo de conflito priorizando a operação shift. Isso permite que o parser interprete corretamente estruturas condicionais com if-then-else, mesmo na presença de possíveis ambiguidades.

Ciclos while e for

As produções referentes aos ciclos while e for são definidas da seguinte forma:

```
for : FOR IDENTIFIER ASSIGN expression to_or_downto expression DO command_list
```

Esta regra define o ciclo for, que permite iterar através de um intervalo de valores. `For`: inicia a construção do ciclo; `IDENTIFIER`: identifica a variável de controlo do ciclo; `ASSIGN`: operador de atribuição para inicializar a variável de controlo; `expression`: expressão que define o valor inicial da variável de controlo; `to_or_downto`: é uma produção auxiliar que determina a direção (TO para incrementeo ou DOWNTO para decremento); `DO`: indica o início do bloco de comandos a serem executados no ciclo; `command_list`: lista de comandos a serem executados em cada iteração do ciclo.

```
while : WHILE expressionBool DO command_list
```

Esta regra define o ciclo while, que executa um bloco de comandos enquanto uma condição for verdadeira. O token `WHILE` inicia a construção do ciclo, seguido pela `expressionBool` que define a condição de continuação do ciclo. O token `DO` indica o início do bloco de comandos a serem executados enquanto a condição for verdadeira, e `command_list` representa a lista de comandos a serem executados.

Operações Binárias

Para definir operações binárias tivemos de dividir em múltiplas regras para não gerar conflitos.

```
<expressionBool> -> <expression>
                  | <expression> <opRel> <expression>

<opRel> -> "=" | "<>" | "<" | "<=" | ">" | ">="

<expression> -> <term>
              | <expression> <opAd> <term>

<opAd> -> "+" | "-" | "AND"

<term> -> <factor>
         | <term> <opMul> <factor>
```

```
<opMul> -> "*" | "/" | "DIV" | "MOD" | "OR"
```

A gramática apresentada organiza as operações em diferentes regras para tratar corretamente a precedência entre operadores aritméticos, booleanos e relacionais. A regra principal é `expressionBool`, responsável por expressões que podem conter comparações. esta produção contém a regra `expression` usada para operações de adição, subtração e conjunção, por sua vez esta regra contém `term`responsável por tratar de operações de multiplicação, divisão e disjunção.

Tratamento da AST

Cada regra de produção na gramática está associada a uma função que constrói o nó apropriado para a AST. Por exemplo:

- *p_program*: Cria um nó Program
- *p_if*: Cria um nó If com condição, bloco then e bloco else
- *p_while*: Cria um nó While com condição e corpo
- *p_for*: Cria um nó For com inicialização, direcção (to/downto), limite e corpo

A precedência de operadores é definida explicitamente para garantir que expressões como a + b * c sejam interpretadas correctamente (multiplicação tem precedência sobre adição).

Análise Semântica

A análise semântica é realizada através da interacção entre a AST e a tabela de símbolos, implementada em symbol_table.py. Este módulo mantém um registo de todas as variáveis declaradas, os seus tipos e posições na stack da máquina virtual.

Verificações Implementadas

- Declaração dupla de variáveis: Erro se uma variável for declarada mais de uma vez
- Verificação de tipos: Em operações e atribuições
- Compatibilidade de tipos: Verificação automática com conversão de inteiro para real quando necessário
- Arrays: Verificação de correto acesso a elementos do array

Tabela de Símbolos

Utilizamos um dicionário `symbols` onde cada entrada é mapeada pelo nome da variável.

- Chave: Nome da variável
- Valor: Um tuplo que pode conter:
  - Para variáveis simples: (tipo, posição na stack)
  - Para arrays: (tipo base, posição na stack, tamanho)

A propriedade `stack_pos` controla a alocação sequencial de posições na stack. Cada declaração simples consome um slot.

A tabela de símbolos armazena:

- Nome da variável

- Tipo (inteiro, real, string, array, etc.)
- Posição na stack da máquina virtual
- Informações adicionais para arrays (tipo base, intervalo, tamanho)

Esta tabela possui métodos para adicionar uma variável guardando os seus dados no dicionário, um método para obter a posição na stack de uma variável e um método de verificação de variáveis não utilizadas.

Ao declarar um array, a tabela calcula o tamanho necessário e reserva apenas uma posição na stack global para armazenar o ponteiro para o bloco de memória alocado na heap.

Operações realizadas:

1. Adição de Variáveis (`add`):
    - Evitar Declarações Duplicadas: Antes de adicionar uma nova variável, o método verifica se ela já existe no dicionário e, caso sim, lança um erro de declaração dupla.
    - Arrays: Se o tipo informado for um dicionário com {'type': 'array', ...}, calcula o tamanho baseado no intervalo definido (ex.: de min a max) e reserva somente um slot na stack para o ponteiro do array. Os parâmetros adicionais, como range e base_type, estão armazenados para possibilitar verificações durante o acesso e a geração de código.
    - Variáveis Simples: Apenas o tipo (convertido para letras minúsculas) e a posição na stack são armazenados.
2. Recuperação da Posição na stack (`get_stack_pos`):
    - Este método permite a consulta da posição armazenada para uma variável.
    - Caso a variável não esteja declarada ou ainda não tenha sido atribuída uma posição válida (posição -1), lança uma exceção ou atualiza a posição conforme necessário, garantindo que sempre exista uma referência válida para o acesso no código gerado.
3. Verificação de Declarações Não Utilizadas:
    - Embora simples, este método varre as declarações e pode imprimir avisos para variáveis que foram declaradas mas nunca usadas, ajudando o compilador a sugerir melhorias e otimizações no uso de memória.

Geração de Código

A geração de código é realizada pelos métodos generate() implementados em cada classe da AST no ficheiro node.py. O código gerado é específico para a máquina virtual alvo, utilizando as suas instruções de stack.

Características do Código Gerado:

- Baseado em stack: A máquina virtual utiliza uma arquitetura em stack
- Alocação de memória: Variáveis globais na stack e arrays na heap
- Manipulação de arrays: Usa instruções específicas como ALLOCN, STOREN, LOADN
- Ajuste de índices: Conversão entre índices Pascal (podem começar em qualquer número) para índices base-0 da máquina virtual

Exemplos de instruções geradas

- Para variáveis simples: PUSHI, STOREG, PUSHG
- Para arrays: ALLOCN, PADD, STOREN, LOADN
- Para operações aritméticas: ADD, SUB, MUL, DIV

- Para estruturas de controlo: JZ, JUMP com etiquetas O código gerado inclui mensagens de erro apropriadas para verificações de tipo em tempo de compilação.

Declaração de variáveis

No nosso compilador, a declaração de variáveis passa por vários passos. Quando o parser reconhece uma variável, envia o nome e o tipo da mesma para a tabela de símbolos e esta armazena esses valores e regista também a posição na stack para os nós depois escreverem os comandos do código-máquina com as posições corretas. Depois nas regras de produção onde o nome da variável volta a aparecer, este é passado como argumento para o nó que usa o nome para procurar os restantes dados relacionados a essa variável e realiza operações sobre a mesma.

Vale também mencionar que a declaração de variáveis é feita de forma dinâmica ao invés de estática, ou seja, estas são declaradas à medida que são usadas e não no início do programa. Uma exceção a esta regra são variáveis em arrays que são declaradas logo no início do programa.

Nós da AST

Como referido na análise sintática, foi necessário associar os valores de cada regra de produção a um nó dedicado. Ou seja, no parser iniciamos o nó, para cada componente e geramos recursivamente o código-máquina. Existem alguns nós que precisam de uma explicação detalhada, sendo eles:

- `BinaryOp`: Como o nome indica, é neste nó que são realizadas as operações binárias do programa. O construtor recebe o código que está à esquerda da operação, o código que está à direita e a operação em si e liga a operação à instrução correspondente no código-máquina. Este nodo também realiza a verificação dos tipos dos argumentos fornecidos, lançando uma exceção se forem incompatíveis para a operação fornecida.

- `If`: Este nó é responsável por representar a estrutura de controlo if-then-else. O construtor recebe a condição, o bloco de comandos do then e o bloco de comandos do else. A geração de código envolve criar tags/etiquetas para saltos condicionais e gerar o código para os blocos correspondentes.

- `For`: Este nó é responsável pelos ciclos for, para tal criamos uma label/tag para o inicio e o fim do ciclo, de seguida o contador é inicializado e verifica-se a condição de paragem. No caso da condição de paragem temos duas opções, quando temos `to` ou `downto`. Para o token `to`, verificamos se i ainda é inferior ao final (`SUP`), se for incrementa-mos o contador i e entramos no corpo do loop. Para o token `downto`, fazemos a verificação inversa (i >= final com a instrução `INF`) e decrementa-mos o contador enquanto a condição for verdadeira.

- `Array`: Este nó representa a declaração e manipulação de arrays. O construtor recebe o nome do array, o tipo base, o intervalo e o tamanho. A geração de código envolve alocar memória na heap para o array (Através de `ALLOCN`) e gerar instruções para acesso e modificação dos elementos, para realizar esta tarefa é colocado o endereço do array no topo da stack com `PUSHG` e aceddido ao seu valor no índice através de `LOADN`. Este nodo também é utilizado para aceder elementos individuais de uma string gerando o código máquina correspondente para obter o código ASCII do caracter especificado (através do comando `CHARAT`).

- `Identifier`: Este nó representa variáveis e identificadores. O construtor recebe o nome da variável e o tipo. Durante a geração de código, este nó é usado para carregar valores na stack da máquina virtual, realizando o comando `PUSHG` sobre o índice da variável obtido da tabela de símbolos.
- `FunctionCall`: Este nó representa chamadas a funções, como writeln e readln. O construtor recebe o nome da função e os argumentos. A geração de código envolve empilhar os argumentos e chamar a função correspondente. Para as funções writeln e write se estas possuirem mais que um argumento a string que vai ser escrita é primeiro concatenada e só depois é chamado o comando WRITES ao invés de cada argumento ser escrito de uma só vez.
- `Literal`: Este nó representa literais (números, strings, booleanos). O construtor recebe o valor literal e o tipo. Durante a geração de código, este nó é usado para empilhar o valor literal na stack da máquina virtual.
- `Assignment`: Este nó representa atribuições de valores a variáveis. O construtor recebe o identificador da variável e a expressão a ser atribuída. A geração de código envolve armazenar o valor na posição correspondente da variável na stack, utilizando o comando `STOREG`.
- `While`: Este nó representa ciclos while. O construtor recebe a condição e o bloco de comandos a ser executado enquanto a condição for verdadeira. A geração de código envolve criar tags/etiquetas para saltos condicionais e gerar o código para o corpo do ciclo.
- `Program`: Este é o nó raiz da AST criada. O construtor recebe o nome do programa, as declarações de variáveis e o bloco de comandos. Durante a geração de código, este nó é responsável por inicializar a stack e devolver o código gerado para o output do programa.

Otimizações

Para otimizar as operações binárias decidimos, no caso de haver apenas literais de ambos os lados da operação, essas operações são realizadas no próprio compilador e apenas o seu resultado é inserido no código-máquina. Mas como podem haver várias operações binárias (Ex: 5 + 5 + 5), introduzimos um campo de value nos nós de `BinaryOp` e `Identifier` e ao gerar o código é verificado se o lado esquerdo e direito têm o campo `value` não vazio para realizar a operação no compilador.

Para além disso, o nosso programa consegue eliminar operações redundantes, como por exemplo x + 0 ou x * 1 apenas gerando o valor de x para o código-máquina.

Outra otimização que realizamos são para as variáveis. O nosso programa, no caso de variáveis não utilizadas, estas nem são declaradas para não ocupar espaço desnecessário na memória.

Testes e Validação

O compilador foi testado com diversos programas Pascal, para além dos exemplos incluídos no enunciado, para verificar a sua correcta funcionalidade.

Teste 1: Operações condicionais em ninho

Para este primeiro teste, testamos como o nosso programa lida com condicionais em ninho (t1.pas).

Como é possível observar pelo output em t1.vm, o compilador reconhece as várias condicionais e consegue distingui-las e o resultado está de acordo com o esperado.

Input: t1.pas | Output: t1.vm

Teste 2: Verificação de operações inválidas

Neste segundo teste verificamos qual seria o output no terminal para uma operação inválida tal como somar um inteiro a uma string. O output é o seguinte:

```
  result := a + b;  { Invalid: cannot add integer and string }
Could not compile program.
```

O compilador encontra o erro e informa o utilizador pelo terminal identificando a operação incorreta e a linha onde o erro ocorre.

Input: t2.pas

Teste 3: Operações binárias entre literais

No terceiro teste, validamos as otimizações realizadas sobre as operações binárias, realizando-as entre literais para confirmar que apenas o resultado dessas operações é gerado no código-máquina. Para além disso, colocamos uma operação redundante para validar a sua eliminação.

Input: t3.pas | Output: t3.vm

Teste 4: Operações aritméticas redundantes

Para testar uma das otimizações implementadas, a eliminação de operações, redundantes testamos os casos onde temos uma variável a somar com 0, ou multiplicações com 0 e com 1.

Para o teste encontrado no ficheiro de input podemos observar que o output encontrado no ficheiro de output o código máquina gerado não executa nenhuma operação ADD ou MUL pois consegue por defeito saber qual o valor que deve fazer push na stack.

Input: t4.pas | Output: t4.vm

Teste 5: Comparações entre literais

Decidimos testar se as comparações entre literais estavam a ser feitas corretamente pois, tal como as restantes operações binárias, estas também são realizadas no próprio compilador e apenas o seu resultado é escrito no código para a VM.

Input: t5.pas | Output: t5.vm

Teste 6: Verificação de variáveis não utilizadas

Realizamos um teste para verificar como o programa trata de variáveis não declaradas como é possível ver em t6.pas, o programa inicializa as variáveis 'a', 'b', 'c','d' e 'e' mas só utiliza 'a', 'b' e 'c', podemos verificar no

terminal o seguinte output

```
Variable 'd' declared but never used.
Variable 'e' declared but never used.
Code compiled in: ../out/t6.vm
```

Podemos verificar que o programa informa o utilizador das variáveis no entanto compila na mesma tendo em conta que o programa não aparente mais nenhum erro grave de lógica ou sintaxe. Além deste pormenor podemos verificar em t6.vm que o programa não alocou memória na stack para as variáveis não aplicadas.

Input: t6.pas | Output: t6.vm

Conclusões

O compilador Pascal implementado demonstra os conceitos fundamentais de compiladores, desde a análise léxica até à geração de código. Os principais desafios e aprendizagens incluíram:

Desafios Enfrentados:

- Tratamento de arrays: Mapeamento entre índices Pascal e endereços na máquina virtual
- Conversão de tipos: Implementação segura de conversões automáticas
- Geração de código para a VM: Compreensão da arquitectura baseada em stack

Melhorias Futuras:

- Suporte a procedimentos e funções definidas pelo utilizador
- Implementação de optimizações de código
- Expansão dos tipos suportados: records, enumerações, etc.
- Mensagens de erro mais detalhadas e amigáveis
- Implementação de mais funções intrínsecas do Pascal

Considerações Finais: O compilador cumpre o seu objectivo de traduzir programas Pascal para o código da máquina virtual alvo, demonstrando os princípios fundamentais de compiladores. A implementação modular permite fácil manutenção e extensão para incluir mais recursos da linguagem Pascal no futuro.